Contribution à la vérication formelle et programmation par contraintes

Hélène Collavizza

Habilitation à Diriger des Recherches

Présentée à

L'Université de Nice-Sophia Antipolis
École doctorale STIC
Spécialité : Informatique

par

Hélène Collavizza

Contribution à la vérification formelle et programmation par contraintes

Soutenue publiquement le jeudi 3 décembre 2009
après avis des rapporteurs

M. **Pascal Van Hentenryck**	Professeur à l'Université de Brown (USA)
M. **Thierry Jéron**	Directeur de Recherches à l'INRIA Rennes
M. **Michaël Rusinowitch**	Directeur de Recherches à l'INRIA Nancy

devant la commission d'examen composée de

Mme **Dominique Borrione**	Professeur à l'Université de Grenoble
M. **Mike Gordon**	Professeur à l'Université de Cambridge (RU)
M. **Thierry Jéron**	Directeur de Recherches à l'INRIA Rennes
M. **Michel Rueher**	Professeur à l'Université de Nice-Sophia Antipolis
M. **Michaël Rusinowitch**	Directeur de Recherches à l'INRIA Nancy
M. **Lakhdar Saïs**	Professeur à l'Université de Lens

Remerciements

Merci à Thierry Jéron, Michaël Rusinowitch et Pascal Van Hentenryck rapporteurs de ce mémoire, ainsi qu'à Dominique Borrione, Mike Gordon, Michel Rueher et Lakhdar Saïs membres du jury qui m'ont fait l'honneur de consacrer une part de leur temps précieux à l'évaluation de ce travail.

Merci à Dominique Borrione, ma directrice de thèse, de m'avoir initiée à la recherche.

Merci à Michel Rueher qui m'a toujours encouragée, merci pour sa patience et pour savoir si bien accepter l'autre tel qu'il est.

Merci à Mike Gordon de m'avoir accueillie dans son équipe à Cambridge. Ces six mois ont été une parenthèse fabuleuse dans ma vie professionnelle où j'ai énormément appris, et de surcroît, avec grand plaisir.

Merci à mes relecteurs, Laurent Arditi, Carine Fédèle, Olivier Ponsini et Michel Rueher, merci à Claude Michel pour son aide.

Merci à tous mes collègues de l'EPU pour la bonne humeur de tous les jours. Un merci particulier à Christophe Coroyer pour son aide efficace dans la gestion des machines. Merci à Anne-Marie Déry, ma voisine de bureau, chez qui une pause m'a toujours revigorée et éclairci les idées. Merci à Marc Gaëtano pour nos années passées à chercher comment enseigner l'algorithmique, et pour tous nos fous rires. Merci à Joël Leroux pour nos discussions sur la spiritualité, teintées de matérialisme primaire. Merci à Audrey Occello dont la seule présence est un modèle. Merci à Jean-Paul Stromboni pour notre collaboration à l'enthousiasme toujours renouvelé pour les projets DeViNT. Merci à Michèle Vermillère pour ses leçons de savoir vivre.

Merci à mes collègues et amis de l'Université, Carine Fédèle, Christine Garcia, Emmanuel Kounalis, Olivier Meste et Laurence Pierre, pour leur présence chaleureuse.

Merci à tous ceux avec qui j'ai partagé la passionante aventure DeViNT, si riche humainement.

Merci à mes élèves, pour la satisfaction que j'ai à leur enseigner, toujours renouvelée.

Merci enfin à mes proches, jardiniers dont l'attention et l'affection quotidienne m'aident à chasser les obstacles et laisser s'épanouir la lumière.

Je dédis ce mémoire à tous ceux qui m'ont guidé sur le chemin.

Table des matières

Avant-propos

Ce mémoire d'habilitation à diriger des recherches présente mes activités de recherche effectuées au *Laboratoire I3S de l'Université de Nice Sophia-Antipolis* depuis Octobre 1992, date de ma nomination comme Maître de Conférences. Cet avant-propos retrace le contexte et la chronologie de mes recherches : j'ai travaillé successivement sur la vérification formelle des microprocesseurs, les contraintes sur domaines continus et la vérification des programmes.

Le thème des recherches de ma thèse de doctorat, effectuée au *Laboratoire d'Informatique de l'Université de Provence* à Marseille était la **vérification formelle des microprocesseurs**. Plus précisément, j'ai proposé une modélisation fonctionnelle du niveau d'abstraction correspondant aux *micro-séquences*, c'est-à-dire le niveau qui correspond aux *étages* dans une architecture *pipeline*. J'ai proposé une méthode de preuve basée sur la *réécriture* afin de vérifier que le niveau *micro-séquences* réalise correctement le niveau d'abstraction supérieur, qui est ici le niveau des *instructions assembleur*.

Lors de ma nomination à Nice en 1992, j'ai intégré la nouvelle équipe que le Professeur Jacques Chazarain était en train de constituer au sein du laboratoire I3S. Cette équipe incluait en particulier le Professeur Emmanuel Kounalis, qui travaillait sur la réécriture et plus précisément sur la définition d'un principe d'induction, les *test sets*. L'objectif de l'équipe était d'explorer différentes techniques de preuve, la vérification des processeurs étant un domaine d'application de ces techniques. J'ai travaillé sur la **vérification formelle des microprocesseurs** pendant cinq ans, en co-encadrant avec Jacques Chazarain le DEA puis la thèse de Laurent Arditi. L'originalité de notre approche a été de proposer une modélisation objet et une approche coopérative combinant différentes techniques de preuve. Pendant cette période, j'ai également travaillé avec Emmanuel Kounalis en co-encadrant le DEA d'Ould Ahmedou Mohamed Lemine sur la généralisation de théorèmes dans les preuves inductives.

A l'issue de cette période, et suite à la réorganisation de l'équipe[1], j'ai changé de thème de recherche en travaillant avec le Professeur Michel Rueher sur les **contraintes en domaines continus**. Cette thématique m'était totalement inconnue, mais il s'est avéré par la suite que mes compétences en mathématiques numériques, ont été des bases précieuses pour me permettre de comprendre et de comparer différentes techniques de résolution de contraintes en domaines continus, qui sont pour certaines proches du calcul numérique. J'ai donc intégré l'équipe de Michel Rueher et co-encadré avec lui le DEA puis la thèse de François Delobel sur la résolution de

[1] après le départ à la retraite de Jacques Chazarain.

contraintes en domaines continus.

Tout en restant dans le thème de la résolution de contraintes, mais cette fois en domaines finis, j'ai ensuite réorienté mes recherches sur la **vérification de programmes**, dans le cadre du projet RNTL DANOCOPS «Détection Automatique de Non Conformités d'un Programme vis-à-vis de ses Spécifications». Des travaux antérieurs d'Arnaud Gotlieb et Michel Rueher avaient montré la pertinence de l'utilisation des contraintes pour représenter un programme et trouver un jeu de test permettant d'atteindre un point dans ce programme. J'ai repris ces idées de fond, et mettant à profit mes connaissances du domaine de la preuve de matériel, j'ai proposé une méthodologie de modélisation avec abstraction booléenne de la partie contrôle du programme, qui combine résolution SAT et résolution de contraintes sur domaines finis. Nous avons ensuite amélioré cette approche en collaboration avec le Professeur Pascal van Hentenryck de l'Université de Brown, en construisant à la volée le système de contraintes numériques lors de l'exploration du graphe de flot de contrôle du programme.

Enfin, j'ai été invitée pendant six mois dans l'équipe du Professeur Mike Gordon à l'Université de Cambridge (Royaume Uni). J'ai pu y explorer une autre méthode de preuve, fondée sur la logique d'ordre supérieur, en utilisant le démonstrateur de théorèmes HOL4. En collaboration avec Mike Gordon, nous avons défini une méthodologie de vérification qui reprend l'exploration à la volée du graphe de flot de contrôle du programme, et qui combine un solveur de contraintes, un solveur SMT et le démonstrateur de théorèmes HOL4. Le point fort de cette approche est qu'elle repose sur une sémantique du langage qui a été formellement définie dans HOL4. Ainsi, l'exécution symbolique d'une instruction dans le chemin en cours d'exploration, est effectuée par réduction automatique de la sémantique opérationnelle définie dans HOL4.

Pendant toutes ces années, *les moteurs de ma démarche scientifique* ont été la curiosité, la volonté de comprendre et de faire coopérer des méthodes issues de domaines de recherche différents, et enfin le souci d'appliquer ces méthodes à des exemples concrets. Je m'attacherai tout au long de ce mémoire à expliquer la démarche commune utilisée et le lien entre les différents thèmes de recherche abordés. Une première constatation est que la démarche que j'ai adoptée dans mes travaux sur la vérification formelle des microprocesseurs et mes travaux sur la vérification formelle de programmes a été guidée par les mêmes objectifs :

- L'effort supplémentaire que le concepteur doit fournir afin d'obtenir une conception sûre doit être *minimisé* en utilisant un langage de *description* et de *spécification* adapté, et si possible familier du concepteur,
- Le processus de vérification par lui-même ne doit pas nécessiter d'intervention du concepteur. L'utilisation de *procédures de décision automatiques* et *adaptées* au niveau d'abstraction du problème est donc privilégiée,
- Si les procédures de décision automatiques ne permettent pas de résoudre le problème alors l'usage *d'assistants de preuve* de haut niveau est envisagé,
- Une *évaluation expérimentale* doit assurer que les méthodes proposées passent à l'échelle afin de valider des applications sinon réelles, mais au moins de taille significative.

Une des spécificités de mon parcours étant d'avoir travaillé sur la vérification de

matériel et puis la vérification de programmes je m'efforcerai d'apporter un éclairage sur les similitudes et différences de ces deux domaines, illustrées par des points précis de mes recherches.

Mes travaux sur la résolution de contraintes en domaines continus peuvent paraître éloignées des deux autres thèmes. Il faut cependant souligner que ces travaux m'ont permis d'appréhender la programmation par contraintes et d'utiliser la résolution de contraintes sur domaines finis comme une procédure de décision pour la vérification des programmes. D'autre part, employer les contraintes sur domaines continus pour vérifier des propriétés de programmes sur les flottants est une perspective incontournable et prometteuse de mes travaux.

Organisation du mémoire

Ce mémoire comporte deux parties : une synthèse de mes travaux de recherche et un curriculum vitæ.

J'ai choisi d'organiser la *synthèse de mes travaux* en présentant d'abord mes travaux sur la *vérification formelle* des processeurs puis des programmes, et en présentant ensuite mes travaux sur les *contraintes en domaines continus*. Cela ne correspond pas à l'ordre chronologique, mais il m'a semblé que cet ordre était plus pertinent pour discourir du lien entre les approches adoptées pour le matériel et pour les programmes.

- L'introduction (chapitre I page 3) définit la problématique de la vérification, en présente les principales méthodes et fait une synthèse des différents thèmes de mes recherches en mettant en perspective leurs points communs.
- Le chapitre «Vérification formelle des processeurs» (chapitre II page 25) présente les travaux effectués durant la thèse de Laurent Arditi. La vérification est automatique dans la plupart des cas ; elle combine des procédures de décision et un assistant de preuves.
- Le chapitre «Vérification formelle des programmes dans un environnement par programmation par contraintes» (chapitre III page 89) présente la méthodologie de vérification de propriétés de programmes qui utilise les contraintes en domaines finis, aussi bien pour représenter le problème que pour le résoudre.
- Le chapitre «Vérification des programmes basée sur la sémantique» (chapitre IV page 163) présente mes travaux actuels sur la vérification formelle des programmes basée sur la sémantique du langage définie formellement en HOL. Ces travaux ont été effectués en collaboration avec le Professeur Mike Gordon lors de mon séjour à Cambridge.
- Le chapitre «Contraintes sur domaines continus» (chapitre V page 209) présente les travaux effectués durant la thèse de François Delobel. Il s'agit d'un travail de synthèse sur les consistances (i.e. propriétés qui permettent de réduire les domaines) ainsi qu'une application originale.
- Le chapitre «Perspectives» (chapitre VI page 259) présente les perspectives de mes travaux sur la vérification des programmes basée sur la sémantique. Il présente également une perspective de mes travaux sur la vérification de

programmes avec les contraintes : la vérification de programmes temps réels, que j'effectue dans le cadre du projet ANR TESTEC «TEst des Systèmes Temps réel Embarqués Critiques», en collaboration avec Michel Rueher et Le-Vinh Nguyen dont je co-encadre la thèse de doctorat.

J'ai choisi d'insérer dans chacun de ces chapitres le contenu des articles publiés qui servent de support à mon propos ainsi que la bibliographie spécifique. Le contenu de ce mémoire reprend en partie le contenu des articles supports, mais j'ai essayé d'apporter certains compléments, et tenté de les rendre plus accessibles à un non spécialiste (ayant abordé plusieurs domaines de recherche, cela m'a semblé nécessaire). En particulier, j'ai formalisé dans le chapitre III la traduction d'un programme en système de contraintes gardées, ce qui n'avait été fait que partiellement, et j'ai commenté au fur et à mesure les points forts ou faibles de l'approche proposée pour justifier les différentes améliorations apportées. Dans le chapitre IV, j'ai expliqué la philosophie spécifique aux assistants de preuve. Enfin, j'ai essayé de rendre le chapitre V plus didactique grâce à des dessins et des exemples détaillés pour éclairer un lecteur non spécialiste de ce domaine.

La partie *curriculum vitæ* détaille de façon classique mes activités d'enseignement, d'administration et de recherche. Un point plus original concerne mes activités de diffusion et de valorisation dans le cadre de la journée DeViNT «Déficients Visuels et Nouvelles Technologies»[2], organisée depuis sept ans par Polytech'Nice Sophia-Antipolis en collaboration avec le CNRS, l'INRIA et l'APEDV (Association des Parents d'Enfants Déficients Visuels). En effet, j'ai développé une synthèse vocale qui est largement diffusée et a été présentée à plusieurs reprises dans des colloques. Cette synthèse vocale a été maintenue et améliorée en collaboration avec mon collègue Jean-Paul Stromboni et grâce à la participation de nombreux étudiants du département informatique. Elle sert de support à des projets d'étudiants en première année de cycle ingénieur (niveau L3) que nous organisons en collaboration avec deux instituts spécialisés. Ces projets constituent une expérience pédagogique innovante où nos élèves sont confrontés à de vrais utilisateurs qui ont de surcroît un handicap. Il est alors crucial de bien cerner leurs attentes afin de réaliser des interfaces adaptées. Les détails sur l'implémentation de cette synthèse vocale, ainsi que sur la diffusion des projets DeViNT dans différents colloques se trouvent dans la partie II au chapitre III page 291.

Conventions *l'emphase* sera utilisée pour mettre en valeur certaines notions, en particulier lors de leur définition.

Lors du développement technique des différents chapitres, les remarques ou choix importants qui ont été faits seront

| mis en évidence grâce à une marge à gauche.

Enfin, j'utiliserai la première personne pour parler de choix ou de contributions qui me sont personnelles. Un travail de recherche ne pouvant avancer que par des

[2] Voir `http://devint.polytech.unice.fr/`

échanges d'idées dans un esprit d'équipe, le «nous» sera par conséquent le plus souvent utilisé.

Première partie

Synthèse des travaux

Chapitre I

Introduction

Ce chapitre introduit les domaines de recherche que j'ai abordés et synthétise les liens entre ces différents domaines. J'ai travaillé sur la vérification formelle des processeurs pendant ma thèse puis jusqu'en 1996 après ma nomination comme Maître de Conférences. J'ai ensuite travaillé sur les contraintes en domaines continus puis sur la vérification des programmes en utilisant les contraintes en domaines finis comme procédure de décision. Il m'a donc semblé intéressant d'introduire mes travaux par une présentation générale du problème de la vérification (de matériel et de logiciel), et de montrer comment les mêmes techniques ont évolué pour s'appliquer à la vérification de matériel d'abord puis à la vérification de programmes. Je ne prétends bien évidemment pas faire un état de l'art ni une comparaison exhaustive entre la vérification de matériel et de logiciel, il s'agit juste de situer mes travaux sur ces deux thèmes et d'en synthétiser les points communs. Pour ce faire, je donne un bref historique avec la vision de quelqu'un qui est passé du matériel au logiciel : je montre en particulier comment les techniques de model checking[1], largement utilisées en vérification de matériel, ont évolué pour permettre aussi la vérification de programmes.

Ce chapitre est organisé de la façon suivante. J'introduis tout d'abord la problématique générale de la vérification en donnant les définitions de base puis en décrivant les méthodes de vérification formelle, en particulier les techniques du model checking. Je présente ensuite mes contributions à la vérification formelle des processeurs et des programmes et mets en perspective les points communs. Dans une dernière section, je décris mes travaux sur la résolution par contraintes sur domaines continus et montre enfin comment ce thème s'articule avec les deux thèmes précédents.

I.1 Problématique de la vérification

Cette section introduit les définitions de base de la vérification. Il ne s'agit pas de définitions formelles et rigoureuses, mais il m'a paru nécessaire de fixer le sens des mots que j'utilise pour parler de ce thème. En effet, chaque communauté de recherche a sa propre culture, et par exemple, la signification du mot "spécification" n'est pas la même quand il s'agit de spécifier un type abstrait `B`, ou quand il s'agit d'énoncer une propriété temporelle que doit vérifier un circuit. Dans le cas de la spécification `B`, un point important est que la spécification doit être *complète*, c'est-à-dire décrire le comportement du programme dans tous ses cas d'utilisation. Dans le deuxième cas, il ne s'agit que d'une propriété *particulière* qui énonce une des caractéristiques du circuit. Pourtant, le processus de raffinement en `B` qui permet de générer un programme *correct*, peut être considéré comme une méthode de *vérification formelle*, de même que la vérification d'une propriété temporelle d'un circuit.

Tout en introduisant le vocabulaire, cette section présente ma vision de la *vérification* que ce soit de matériel ou de logiciel. Je ne m'étendrai pas sur l'intérêt de la vérification, tant il me paraît évident que concevoir du matériel ou du logiciel *correct* est une nécessité, que ce soit sur le plan de la sécurité (e.g. logiciel ou matériel embarqué en avionique) ou de l'impact économique.

[1] J'ai choisi de ne pas traduire ce terme et de le noter avec une fonte sans serif.

I.1.1 Définitions

Pour définir le problème de la vérification, j'ai choisi la définition de Sommerville [30] qui est simple, pragmatique et assez large pour s'appliquer aussi bien au cas du matériel que du logiciel.

Vérification la *vérification* est la réponse à la question

"Are we building the product right ?".

Cette définition sous-entend trois éléments de base à la vérification : "right" met en évidence une notion de *référence* qui énonce ce qui doit être correct. "building the product" énonce un processus de conception : une partie du produit est en cours de *réalisation*. Enfin "Are we" met en évidence le besoin d'une méthode pour s'assurer que la réalisation en cours est correcte.

Spécification une *spécification* est une *propriété* qui sert de *référence* pour la vérification : c'est la propriété à vérifier pour le produit en cours de réalisation. Une spécification peut contenir des erreurs ou être incomplète. Une même propriété peut servir de *spécification* à une étape donnée, et être la *réalisation* qui doit vérifier une spécification pendant une autre étape du processus de vérification. Par exemple, la description d'un circuit au niveau *transfert de registres* est la spécification du circuit vu au niveau *portes logiques*, mais c'est la réalisation du circuit vu au niveau *instructions assembleur*.

Test de logiciel le *test* est la méthode de *vérification* la plus largement employée pour le logiciel : tout programmeur, de l'étudiant débutant à l'ingénieur confirmé, effectue des tests pour essayer de se convaincre que le logiciel qu'il est en train d'écrire réalise bien la fonction souhaitée. Ces tests sont généralement écrits "à la main" en choisissant des valeurs pertinentes pour l'application considérée, et en vérifiant que ce qui est calculé pour ces valeurs correspond bien à ce qu'on attend. Des outils comme Junit pour java [5] permettent de mécaniser la vérification du résultat du test, et facilitent ainsi les tests de non régression. De nombreux outils permettent de générer de façon automatique ou semi-automatique des jeux de test, en imposant certains critères, comme la couverture de chemins (voir [6] pour une introduction détaillée aux techniques de test).

Simulation de matériel la *simulation* est la méthode de *vérification de matériel* la plus largement employée dans l'industrie. Des outils de CAO permettent de décrire des circuits dans un *langage de description de matériel*, VHDL et VERILOG étant parmi les plus utilisés, et de *simuler* leur fonctionnement. Cela signifie que l'on va exécuter un *modèle* du circuit, pour un certain nombre de *pas* de simulation (i.e. top d'horloge), et pour certaines valeurs d'entrée. Comme pour le test logiciel, des méthodes automatiques permettent de générer des jeux de test en assurant certains critères comme par exemple découvrir le plus grand nombre de "stuck-at" possible, c'est-à-dire de fils qui restent toujours à 1 ou à 0.

De façon synthétique, on peut dire que le *test de logiciel* et la *simulation de matériel* sont des méthodes de *vérification* qui valident le système pour un ensemble de valeurs et répondent donc à la question :

> Un ensemble de valeurs d'entrée appliqué à la réalisation satisfait-il la spécification ?

Vérification formelle la *vérification formelle* est la réponse à la question :

> Est-ce que la réalisation satisfait la spécification quelles que soient les valeurs d'entrée ?

La *vérification formelle* introduit donc un quantificateur universel. Il apparaît donc clairement que ce problème ne peut être résolu par des méthodes probabilistes, mais que cela nécessite au contraire d'utiliser des méthodes formelles basées sur des mathématiques. Il s'agira de trouver un *modèle mathématique* permettant d'exprimer aussi bien la réalisation que la spécification, et fournissant des méthodes de preuve adaptées.

Notons que même si la vérification formelle apporte une réponse plus fiable que la vérification, ce n'est pas une panacée. Avra Cohn en explique les raisons fondamentales dans un article souvent cité en référence dans la communauté de vérification de matériel (voir [20]) :

> "Neither an intended behaviour nor a physical chip is an object to which the word "proof" meaningfully applies. Both an intention and a chip may themselves be inadequately represented in formal language, and this is not itself verifiable."

Vérification formelle complète ou incomplète La vérification formelle est *complète* si la vérification est effectuée en explorant de façon exhaustive tous les états successifs possibles du système. La vérification est *incomplète* si la vérification ne considère que des séquences d'état de longueur *bornée*. Pour la vérification de circuits, cela signifie que l'on va vérifier la propriété pour un nombre de cycles borné ; pour la vérification de programmes, cela signifie que l'on borne le nombre de passages dans les boucles. Par exemple, le model checking ou l'induction en utilisant un démonstrateur de théorèmes sont des méthodes formelles complètes. Par contre, le bounded model checking est une méthode de vérification formelle incomplète (voir section I.2).

Vérification formelle partielle ou totale des programmes la vérification *partielle* consiste à vérifier le programme en supposant que le programme termine, tandis que la *vérification totale* nécessite de montrer en plus que le programme termine. Notons que ce problème de terminaison n'a pas de sens dans la vérification de matériel puisqu'un circuit a pour vocation de fonctionner indéfiniment.

Procédure de décision une *procédure de décision* est un algorithme qui termine[2] en répondant oui ou non à un problème de décision (qui est par conséquent

[2]en un temps éventuellement très long, et donc inacceptable en pratique

décidable). Les procédures de décision sont liées à une théorie, c'est-à-dire à une logique sous-jacente. Par exemple, un solveur SAT est une procédure de décision pour la théorie du calcul propositionnel qui est décidable et NP-complète. La résolution de contraintes sur domaines finis (voir chapitre III section III.2.1) est une procédure de décision sur la théorie des entiers bornés.

Démonstrateur de théorèmes un *démonstrateur de théorèmes* a pour vocation de traiter le cas des théories *indécidables* comme celle des entiers par exemple. Il s'agit en général d'un outil interactif (i.e. assistant de preuves) qui va utiliser des procédures de décision pour les théories décidables, et un principe de définitions associé à des *règles d'inférence* pour dériver de nouvelles propriétés vraies à partir de propriétés déjà démontrées. Effectuer une preuve avec un démonstrateur de théorèmes peut nécessiter une intervention humaine. Par exemple, *l'induction* est une règle d'inférence classique pour la théorie des entiers naturels, qui nécessite très souvent l'intervention de l'utilisateur pour indiquer sur quelle variable effectuer l'induction et quelle hypothèse d'induction utiliser.

I.1.2 Contexte de mes travaux

Dans mes travaux sur la vérification de matériel et de logiciel, je me suis intéressée uniquement à la *vérification formelle*. J'ai privilégié dans les deux cas *l'automatisation* de la vérification en utilisant des *procédures de décision* automatiques. Pour la vérification des programmes, la vérification est *partielle* puisque je ne m'intéresse pas au problème de terminaison. Elle est aussi *incomplète* puisque les boucles sont dépliées. Par contre, pour la vérification des processeurs, notre chaîne de vérification permet la vérification *complète* dans le cas des instructions assembleur qui contiennent des boucles, grâce à l'emploi d'un assistant de preuves. Enfin, la *spécification* consiste en la représentation logique des instructions assembleur pour la vérification des processeurs, et en un couple (pré-condition, post-condition) pour la vérification des programmes.

I.2 Les méthodes de vérification formelle

La *vérification formelle* concerne la vérification du système *quelles que soient les valeurs d'entrée*. Le défi de cette approche est de représenter les domaines d'entrée dans leur totalité afin de raisonner pour toutes les valeurs possibles. Pour cela, deux grandes approches se dégagent : les *méthodes déductives* et les *méthodes automatiques*.

I.2.1 Méthodes déductives

Pour les méthodes déductives, il s'agit d'utiliser directement les théories définies dans un démonstrateur de théorèmes. L'ensemble des entrées du système est donc représenté par un ensemble de variables typées avec une théorie prédéfinie (e.g. théorie des entiers, théorie des listes, ...). Le système est décrit directement dans la logique sous-jacente sous la forme d'un ensemble de définitions et des règles

d'inférences permettent de raisonner sur ces définitions afin de déduire la propriété à vérifier. Le démonstrateur de théorèmes mécanise la preuve sur trois aspects :

- *vérifier que la déduction proposée est correcte*, c'est-à-dire étant donné une formule à démontrer, vérifier que la règle d'inférence choisie s'applique, l'appliquer et en déduire les nouvelles formules vraies ou qui restent à prouver,
- *assister l'utilisateur dans la construction de la preuve* en proposant des règles d'inférence à appliquer,
- *fournir des procédures de décision* pour prouver certaines formules de façon automatique.

L'atout majeur de cette approche est sa rigueur puisque toutes les définitions sont écrites au sein d'un démonstrateur de théorèmes dans lequel toutes les théories ont été prouvées au préalable et où un certain nombre de vérifications sont effectuées de façon automatique sur les définitions (e.g. s'assurer qu'il existe un ordre de réduction montrant qu'une définition récursive décroît). Le second atout est que dans le cas d'une théorie indécidable, la vérification peut quand même être effectuée, si besoin avec l'aide de l'utilisateur qui intervient en donnant des lemmes et en précisant quelle règle d'inférence appliquer. Le défaut est l'effort requis pour décrire le système, qui doit être en grande partie reconduit pour vérifier une autre propriété.

Les méthodes déductives ont été largement utilisées pour la vérification de matériel, en particulier à ses débuts comme expliqué dans l'état de l'art du chapitre II page 25. En vérification de programmes, la tendance actuelle est d'utiliser les méthodes déductives de façon combinée avec des approches automatiques [25, 2, 11]. L'idée est alors de générer des obligations de preuve vers un assistant de preuve mais aussi un ensemble de procédures de décision automatiques. L'assistant de preuve sert à suppléer les procédures de décision quand la preuve diverge ; on peut alors fournir des lemmes ou définir de nouvelles tactiques "à la main". Ces méthodes requérant bien souvent une trop grande expertise et intervention humaine, elles sont peu utilisées dans l'industrie.

I.2.2 Méthodes automatiques

Pour les méthodes de vérification formelle *automatiques*, l'hypothèse de départ est que les domaines à explorer sont finis (bien que très grands). Il est donc possible, même si cela est coûteux, de vérifier la propriété en énumérant toutes les valeurs possibles. Évidemment, cela nécessite la mise en place d'algorithmes et de représentations efficaces pour retarder l'explosion combinatoire. L'avantage de telles méthodes est qu'elles sont entièrement automatiques mais l'inconvénient est que l'utilisateur ne peut pas intervenir dans le cas où la complexité de la combinaison des entrées rend le calcul impossible en pratique. Je présente ci-dessous la méthode du model checking qui est la plus populaire et la plus utilisée dans l'industrie, et que l'on retrouve aussi bien en vérification de matériel que de logiciel. La présentation est chronologique et montre comment le model checking a évolué avec l'apparition de techniques efficaces de représentation des données (BDDs) et des procédures de décision performantes (solveur SAT) .

Model checking Les premiers travaux sur le model checking sont attribués à Edmund Clarke [14] et Joseph Sifakis [29] qui ont reçu le prix Turing en 2007 pour leurs contributions dans ce domaine. Le model checking a été défini à l'origine pour vérifier des propriétés des systèmes concurrents. Le système à vérifier est représenté par un *système de transition d'états* et la propriété à vérifier est exprimée en *logique temporelle*. La logique temporelle [7] combine la logique combinatoire et des opérateurs temporels et permet d'exprimer des propriétés du type "chaque requête devra être acquittée", ou bien d'exprimer qu'une assertion est un invariant, à savoir qu'elle est toujours vraie. Les formules sont définies récursivement à partir des formules du calcul propositionnel augmenté d'opérateurs temporels. Par exemple, si f_1 et f_2 sont des formules CTL "Computation Tree Logic" [6], alors $\mathbf{EX}f_1$ (E pour "Exists" et X pour "neXt") est la formule CTL vraie dans l'état s si et seulement si f_1 est vraie pour un des états successeurs de s, et $\mathbf{E}f_1\mathbf{U}f_2$ ($\mathbf{U}$ pour "until") est la formule CTL vraie dans l'état s_0 si et seulement si il existe un chemin $s_0, s_1, ..., s_i, ..., s_n$ et il existe $i \geq 0$ tel que f_1 est vraie dans les états s_0 à s_{i-1} et f_2 est vraie dans l'état s_i.

L'algorithme le plus simple (i.e. model checking explicite) pour vérifier une formule en logique temporelle consiste à effectuer un parcours explicite du graphe de transitions d'état. Les sommets du graphe sont étiquetés successivement avec les sous-formules de la formule f à prouver. A chaque étape, des règles permettent de savoir comment étiqueter les sous-formules de taille i à partir de celles de taille $i-1$. Par exemple, si $g = \neg g_1$, tous les états qui ne sont pas étiquetés par g_1 sont étiquetés par g, et pour l'opérateur temporel $\mathbf{U}$, l'étiquetage se ramène à un problème d'atteignabilité dans le graphe. Le model checking explicite des formules CTL est décrit plus en détail dans l'annexe VII.1 page 268.

Quand l'ensemble des états S est *fini*, le problème du model checking d'une formule f est décidable, mais dans le pire des cas, la vérification par model checking explicite nécessite un parcours du graphe pour chaque sous-formule de f et sa complexité est donc $O(|f| * (|S| + |R|)$ où R est le nombre de transitions d'états. En pratique, cet algorithme n'est pas toujours efficace, en particulier pour vérifier des propriétés de circuits séquentiels, et c'est avec l'introduction du model checking *symbolique* que cette méthode a vraiment pris son essor.

Model checking **symbolique** Le model checking *symbolique* utilise une représentation symbolique des états. L'idée de base est d'utiliser des *vecteurs de bits* pour représenter les états (i.e. un bit par variable booléenne), et d'utiliser des fonctions sur les vecteurs de bits pour représenter les *ensembles* d'états [12, 11]. Par exemple, avec les variables $\{v_1, v_2, v_3\}$, l'ensemble d'états $\{(1,1,0), (1,0,0)\}$ peut être représenté par la fonction booléenne $v_1 \wedge \neg v_3$. Les fonctions de transition d'états sont elles aussi représentées par des fonctions booléennes[3].

Le point clef de l'efficacité de la méthode de model checking symbolique est de stocker les formules booléennes dans une structure de données adéquate, basée

[3]On peut faire un parallèle évident entre cette représentation et la façon de coder la partie contrôle d'un processeur par une machine de Moore. On peut associer à une telle machine un circuit générique décomposé en deux parties : une fonction booléenne qui calcule les sorties en fonction de l'état courant, et une fonction booléenne qui calcule le prochain état en fonction de l'état courant et des entrées [51].

sur les BDDs (Binary Decision Diagrams) [1, 3, 4] (cette structure de données est décrite dans l'annexe VII.1.3). Ainsi, les fonctions caractéristiques des états et les fonctions de transition d'états sont représentées par des BDDs. De même, les règles d'étiquetage des états et le problème d'atteignabilité sont directement implémentées par des opérations sur les BDDs [11].

Le model checking symbolique a été largement utilisé pour la vérification de matériel, ou des protocoles. Même si cette technique permet de vérifier des systèmes de grande taille (cf les fameux "10^{20} states and beyond" [12]), le problème de l'explosion combinatoire est toujours présent, et les recherches visent à limiter ce problème en améliorant par exemple la gestion des BDDs, ou en travaillant par composition.

Le lecteur intéressé par une revue plus complète des techniques de model checking pour la vérification de matériel peut se référer à [32] ; il trouvera dans [11] tous les détails sur le model checking symbolique, et dans [22] un tutoriel très didactique sur ces méthodes.

Abstraction des prédicats Le model checking symbolique utilise une représentation efficace des formules booléennes. Une façon orthogonale pour accroître l'efficacité est de *simplifier* le modèle en cours de vérification en prenant une *abstraction*. Ce principe a été utilisé en vérification de matériel (propriétés en logique temporelle de programmes assembleur) [16] puis de logiciel [14]. Intuitivement, l'abstraction revient à simplifier ou enlever des détails ou des composants entiers qui ne sont pas nécessaires à la propriété en cours de vérification. Le gain est évident puisque la propriété à prouver est plus simple ; mais cela a aussi un coût puisque vérifier un modèle plus abstrait peut éventuellement amener à de faux résultats.

Dans le cas de la vérification des programmes, l'abstraction de prédicats permet d'abstraire un programme en un programme booléen séquentiel, pour lequel le problème d'atteignabilité est décidable. Cependant, si les prédicats réduisent l'espace de recherche et donc permettent de déterminer plus facilement la satisfaisabilité de la formule sur le système abstrait, ils ont l'inconvénient de générer des "spurious counter-examples" que je nommerai "contre-exemples fallacieux". En effet, l'introduction des prédicats induit une perte de sémantique et la formule abstraite peut être satisfaite alors que la formule concrète ne l'est pas. Le processus de vérification par model checking et abstraction de prédicats est donc le plus souvent mis en jeu en vérification de programmes sous la forme d'un processus itératif, nommé CEGAR dans la littérature pour "Counter-Example Guided Abstraction Refinement" [17, 14, 15].

Ce processus est en quatre étapes itérées (éventuellement une infinité de fois) :

1. *Étape d'abstraction :* choisir les prédicats et calculer la formule abstraite,
2. *Étape de vérification :* tester la satisfaisabilité de la formule abstraite. Si elle est n'est pas satisfiable, alors la formule concrète ne l'est pas non plus. Sinon, soit τ un contre-exemple.
3. *Étape de simulation :* vérifier que τ est effectif sur le modèle concret. Si oui, stopper avec détection de l'erreur τ. Sinon, τ est un contre-exemple fallacieux,
4. *Étape de raffinement :* si τ est fallacieux, raffiner l'abstraction pour éliminer τ.

Notons que *l'étape d'abstraction* nécessite d'une part de déterminer un ensemble de prédicats, mais nécessite aussi l'utilisation de procédures de décision pour transformer le modèle concret en un modèle abstrait puisqu'il faut, pour chaque état, être capable de décider s'il satisfait ou non le prédicat.

Ce processus de raffinement guidé par les contre-exemples est implémenté dans le model-checker NuSMV [17]. Pour la vérification de programmes, il est au cœur du projet SLAM [4] et il est utilisé dans de nombreux autres outils (voir le tableau récapitulatif dans l'état de l'art de D'Silva, Kroening et Weissenbacher [23]). Évidemment, la difficulté majeure de l'approche consiste à construire (de façon automatique) les prédicats les plus précis, qui génèrent le moins de contre-exemples fallacieux.

Bounded model checking Toutes les techniques de model checking ci-dessus sont des méthodes complètes : on fait l'hypothèse que l'ensemble des états est borné et l'on peut donc énumérer tous les chemins dans ce graphe de transition d'états (via une représentation efficace des états et une abstraction). Avec l'émergence des solveurs SAT fin des années 90, dont l'implémentation efficace permet de tester la satisfaisabilité d'une formule booléenne contenant des milliers de variables, une autre approche a été mise en application : le bounded model checking [7].

L'idée est la suivante. Plutôt que d'explorer de façon dynamique l'espace des états atteignables, il s'agit de construire une formule booléenne qui représente la conjonction de la négation de la formule à prouver et les chemins d'une longueur bornée k atteignables dans le graphe de transition d'états. De façon concrète, cela signifie que dans un programme, une boucle *while* sera *dépliée* c'est-à-dire remplacée par la succession de $if\ else$ correspondante : on itère k fois la substitution de $while(c)B;$ par $if(c)B; while(c)B;$. Dans le cas des circuits, c'est la relation de transition d'états qui est dépliée : si l'état E' dépend de l'état E alors l'expression de E' est expansée en faisant apparaître les k états intermédiaires $E_0, E_1, ..., E_{k-1}$. Une fois la formule du programme ou du circuit "dépliée" jusqu'à la longueur k, la formule est passée au solveur SAT qui teste sa satisfaisabilité. Si elle est satisfiable, alors on a trouvé un cas d'erreur, sinon, on itère le processus avec une borne k plus grande. Contrairement au model checking, le bounded model checking n'est pas une méthode formelle complète. Cependant, c'est une des méthodes formelles les plus utilisées dans l'industrie, que ce soit pour la vérification de matériel que de logiciel puisqu'elle a le mérite d'être automatique et applicable à des exemples de taille réelle. De plus, les contre-exemples fournis sont de longueur minimale dans le sens qu'ils sont trouvés par dépliages croissants. Ceci aide à la correction du système en cours de vérification.

I.2.3 Méthodes de vérification formelles utilisées dans mes travaux

Dans mes travaux sur la vérification formelle des processeurs et des programmes, le cœur de la méthodologie de vérification est d'effectuer un model checking en calculant les états par *exécution symbolique* [27]. Le graphe de transition d'états n'est pas construit de façon explicite, mais au contraire, les transitions d'états sont effectuées par calcul d'un état symbolique qui est fonction de l'état précédent. Par exemple, pour les programmes, l'état est l'ensemble des variables du programme, et pour les

processeurs, l'état est l'ensemble des registres et mémoires accessibles au programmeur assembleur. Pour obtenir des états plus simples, le calcul d'état s'effectue par *calcul formel* c'est-à-dire en appliquant des règles de simplification formelles dans un ordre bien établi.

Les similitudes avec les différentes variantes du model checking sont les suivantes :

1. Pour la vérification des processeurs, nous avons utilisé une structure de données dérivée des BDDs, les *BMDs (Binary Moment Diagrams) pour représenter les états (voir chapitre II section II.2.3 page 34). Cette structure offre une représentation compacte des expressions arithmétiques sur les vecteurs de bits et permet donc de retarder l'explosion combinatoire lors de la construction des états symboliques. Cette idée est similaire à celle du model checking *symbolique*, puisqu'il s'agit dans les deux cas de représenter les états de façon efficace en espace.
2. Pour la vérification des programmes, nous avons utilisé du bounded model checking. Cependant, contrairement aux approches classiques, nous n'avons pas utilisé de solveur SAT pour effectuer la vérification mais choisi de tirer parti de la programmation par contraintes en travaillant directement sur les entiers (voir chapitre III sections III.3 et III.4).
3. Dans notre première approche sur la vérification de programmes, nous avons utilisé une abstraction booléenne de la partie contrôle du programme en résolvant un système hybride qui combine booléens pour la partie contrôle et entiers pour la partie opérative (voir chapitre III section III.3 page 99). Cette approche a des similitudes avec *l'abstraction de prédicats* dans le model checking puisque nous abstrayons une partie du système pour diminuer l'espace de recherche. Toutefois, comme les contraintes permettent de résoudre un système contenant à la fois des booléens et des entiers, notre méthode a l'avantage de générer peu de contre-exemples fallacieux.

I.3 Contributions à la vérification formelle des processeurs et des programmes

I.3.1 Vérification formelle des processeurs

Le contexte La vérification formelle des processeurs consiste à montrer que le jeu d'instructions assembleur d'un processeur est correctement réalisé par une implémentation au niveau *transfert de registres*. Les caractéristiques d'une telle vérification sont les suivantes :

- la preuve consiste à prouver la commutativité de diagrammes de transition entre des niveaux de description successifs : exécuter une étape du niveau i doit donner le même état que l'abstraction de l'état qui est obtenu après exécution de n étapes au niveau plus concret $i-1$;
- la description du niveau instruction assembleur utilise des types de données de haut niveau comme les entiers machine ;
- les itérations doivent être prises en considération puisque les instructions assembleur peuvent contenir des boucles comme par exemple l'instruction “REP

MOVSB" de la famille du 8086 qui recopie une zone mémoire vers une autre.

Notons que le problème énoncé ci-dessus était d'actualité à l'époque où nous avons effectué ces recherches (début des années 90). Cependant, avec la complexité croissante des processeurs, et en particulier les mécanismes de parallélisme (e.g. pipeline, exécution dans le désordre,...) ainsi que la multiplicité des composants de haut niveau qui constituent un processeur actuel (e.g. Memory Management Unit, floating point execution unit, ...) il est maintenant illusoire de vérifier formellement un processeur dans sa totalité. Par contre, des méthodes formelles sont utilisées pour vérifier des composants de bas niveau ou pour vérifier des propriétés sur l'exécution parallèle, comme par exemple la gestion du registre de ré-ordonnancement dans les architectures avec exécution dans le désordre (voir la conclusion du chapitre II).

Contributions Notre méthodologie de vérification est basée sur un calcul d'états par exécution symbolique et privilégie les méthodes automatiques et complètes. Les contributions principales sont les suivantes :

1. La vérification est *automatique* dans la plupart des cas et intègre des techniques de vérification *hybrides*. Un moteur d'exécution symbolique et un système de calcul formel spécifique permettent de calculer de façon efficace les transitions d'état. Les *BMDs permettent de retarder l'explosion combinatoire des expressions obtenues dans le cas d'instructions implémentées par une boucle de longueur fixe. Enfin, l'assistant de preuve *Coq* sert à effectuer une preuve inductive dans le cas d'instructions implémentées par une boucle dont la longueur n'est pas fixée par la taille des données.
2. L'approche a été appliquée à de *nombreux processeurs de taille significative*. En particulier, nous avons vérifié de façon automatique des processeurs qui avaient été vérifiés de façon manuelle, et nous avons vérifié un processeur réel conçu par le CNET à Grenoble. De plus, nous avons vérifié des instructions de boucle, ce qui n'avait pas été fait jusqu'alors.
3. Enfin, soucieux de minimiser l'effort requis par l'utilisateur pour effectuer une vérification formelle plutôt qu'une simulation, nous avons proposé un *modèle de spécification* générique, orienté objet qui permet de décrire les niveaux d'abstraction et les preuves elles-mêmes. A partir de ce modèle, et grâce à l'étude de cas d'un processeur de taille significative décrit en VHDL [8], nous avons identifié un sous-ensemble de VHDL que nous sommes capables de traiter.

Les points clefs de cette approche se résument donc ainsi : automatisme, simplicité d'utilisation et mise en application sur des exemples significatifs. Ces travaux ont été réalisés en collaboration avec Jacques Chazarain ; ils sont décrits en détail dans le chapitre II page 25.

I.3.2 Vérification formelle des programmes par programmation par contraintes

Le contexte Mes travaux concernent la *vérification formelle partielle et incomplète* de triplets de Hoare $(pre, prog, post)$ où $prog$ est le programme, pre est une pré-condition (propriété qui doit être vraie *avant* exécution du programme) et $post$ est une post-condition (propriété qui doit être vraie *après* exécution du programme) en utilisant la *programmation par contraintes sur domaines finis* comme procédure de décision. L'objectif initial de ces travaux était d'évaluer comment le cadre uniforme de la programmation par contraintes, qui permet le traitement aussi bien des booléens, des entiers que des nombres flottants, pouvait offrir de meilleures performances que les approches classiques de model checking basées sur une traduction booléenne du programme.

Les systèmes de contraintes et leurs mécanismes de résolution sont introduits brièvement dans la section I.4 de ce chapitre. La résolution de contraintes en domaines finis sera détaillée section III.2.1 page 94 ; il suffit pour l'instant de la voir comme une procédure de décision pour tester la satisfaisabilité d'un ensemble d'-expressions construites à partir des opérateurs usuels sur les booléens et un sous-ensemble fini des entiers, et contenant des variables dont le domaine est *discret* et *borné*. Nous utilisons aussi ici la notion de contraintes *gardées* dont le mécanisme de résolution est détaillé page 94. De façon grossière, il s'agit de contraintes qui sont posées seulement quand une condition (la garde) est satisfaite.

Nous avons choisi d'effectuer un bounded model checking en utilisant les contraintes pour représenter la pré-condition et les exécutions possibles du programme et en montrant que la négation de la post-condition est inconsistante avec le système de contraintes construit. Plus précisément :

- pre, $prog$ et $\neg post$ sont traduits en un ensemble de contraintes C_{pre}, C_{prog} et C_{not_post}. Chaque variable du programme (paramètres et variables locales) est associée à une variable du système de contraintes avec comme domaine initial $[-2^{f-1}, 2^{f-1} - 1]$ où f est le format des entiers. Chaque instruction du programme est traduite en une ou plusieurs contraintes qui restreignent les valeurs que peuvent prendre ces variables. Les instructions conditionnelles sont traduites en contraintes gardées, dont la propagation dépend de la validité d'une garde. Les instructions de boucle sont traduites en un ensemble de contraintes gardées qui correspond à un nombre maximum max de passages dans la boucle.
- le système de contraintes $C = C_{pre} \wedge C_{prog} \wedge C_{not\_post}$ est résolu.
- si C a une solution, alors cette solution est un cas d'erreur d'utilisation puisqu'elle satisfait les contraintes de la pré-condition et du programme mais viole les contraintes de la post-condition. Cela signifie qu'il existe une configuration des variables du programme qui correspond à un cas d'exécution du programme qui ne satisfait pas la post-condition.
- si C n'a pas de solution, alors le programme est correct pour au plus max passages dans les boucles.

La puissance de cette méthode dépend de deux points clefs : résoudre efficacement les contraintes gardées qui sont associées aux instructions de contrôle et résoudre

efficacement un système de contraintes où les domaines des variables sont très grands (i.e. entiers codables en machine) alors que les applications usuelles des contraintes concernent des domaines plus petits comme par exemple l'ensemble des professeurs pour un problème d'emploi du temps.

Contributions Dans ce contexte, nous avons proposé deux approches. La première approche combine contraintes booléennes pour la partie contrôle du programme et contraintes sur les entiers pour la partie opérative. Cela permet de résoudre efficacement les contraintes gardées puisque les gardes sont représentées par des variables booléennes qui sont instanciées au plus vite par l'étape de recherche[4]. L'idée est similaire au model checking avec abstraction de prédicats, mais l'atout majeur est que les booléens et les entiers sont gérés conjointement ce qui réduit le nombre de contre-exemples fallacieux.

La deuxième approche repose sur une exécution symbolique du programme, en coupant à la volée les chemins qui ne sont pas exécutables. Les instructions conditionnelles ne sont plus traduites de façon statique par des contraintes gardées, mais au contraire, leurs conditions sont résolues de façon dynamique en testant leur consistance par rapport au système de contraintes courant (qui représente de façon symbolique un chemin d'exécution du programme). Cette exécution symbolique combine un solveur booléen, un solveur MIP (Mixed Integer Progamming) pour les expressions linéaires et un solveur de contraintes en domaines finis. Le solveur linéaire est très rapide et ses temps de calcul sont indépendants de la taille des domaines.

Les points forts de ces travaux se résument ainsi :
- Nous avons proposé et implémenté une méthode *automatique* de bounded model checking basée sur la programmation par contraintes. Il s'agit à notre connaissance de la première mise en œuvre effective d'une telle approche,
- Nous avons montré que la programmation par contraintes est un paradigme *pertinent* pour la vérification formelle des programmes,
- Nous avons mis en évidence *la puissance des résolutions hybrides* qui combinent plusieurs solveurs,
- Nous avons appliqué notre méthodologie à de *nombreux exemples de taille significative*,
- L'approche par exécution symbolique du programme est très efficace et se *compare favorablement* aux autres approches basées sur le bounded model checking.

Ces travaux sont détaillés au chapitre III. Les principes de la résolution de contraintes en domaines finis sont présentés section III.2.1 page 94. L'approche par abstraction booléenne est décrite section III.3 page 99 et l'approche par exécution symbolique section III.4 page 112. Ces travaux ont été réalisés en collaboration avec Michel Rueher et en partie avec Pascal Van Hentenryck de l'Université de Brown.

[4] L'étape de recherche consiste à choisir une variable et une valeur. Cette étape est combinée avec une étape de *filtrage* qui réduit les domaines des variables. Une des heuristiques de l'étape de recherche consiste à sélectionner les variables qui ont le plus petit domaine, booléen ici. Ceci est détaillé section I.4

I.3.3 Vérification formelle des programmes basée sur la sémantique

Le contexte L'approche précédente, et de manière plus générale, les approches automatiques par bounded model checking n'explicitent pas la sémantique du langage de programmation. Au contraire, la sémantique intervient de façon implicite dans la traduction des instructions vers la syntaxe de la procédure de décision ciblée. Mes travaux en cours sur la vérification des programmes basée sur la sémantique visent à combler cette lacune : l'étape préliminaire à la vérification d'un programme est d'avoir défini formellement sa sémantique.

Contributions En collaboration avec le professeur Mike Gordon, nous avons tout d'abord repris l'approche par exécution symbolique en basant chaque étape d'exécution sur une sémantique opérationnelle du langage définie formellement en logique d'ordre supérieur. D'autre part, nous avons formalisé l'exécution symbolique comme une génération de plus forte post-condition. L'objectif de ces travaux est d'intégrer le bounded model checking et les preuves complètes par génération de plus forte post-condition dans le même cadre formel basé sur la sémantique du langage. Ces travaux sont détaillés au chapitre IV.

I.3.4 Points communs des travaux

Les points communs qui se dégagent de ces travaux peuvent se synthétiser ainsi : la même exigence de minimiser l'effort supplémentaire requis par la vérification formelle et une mise en œuvre de techniques similaires pour satisfaire à cette exigence.

- Minimiser l'effort requis par le concepteur du système en cours de vérification est une condition nécessaire pour qu'il accepte de changer ses habitudes : passer des techniques de simulation ou de génération de jeux de tests à la vérification formelle. Il faut donc partir d'un langage de spécification usuel et privilégier les méthodes automatiques qui ne requièrent pas d'intervention de l'utilisateur. Si une intervention est nécessaire, elle doit être guidée et compréhensible,
- Pour cela, les méthodes formelles incomplètes offrent l'avantage d'être entièrement automatiques. Même s'il s'agit principalement d'une recherche d'erreurs, la différence avec les méthodes de test est néanmoins fondamentale : la vérification est formelle et donc valide pour toutes les données d'entrée. L'utilisateur a l'assurance que la vérification couvre entièrement le système qui a été déplié,
- Selon le système en cours de vérification, et la classe plus spécifique auquel il appartient, les types de données et les paradigmes nécessaires à sa vérification sont différents. De plus, un même système peut inclure des sous-systèmes qui font intervenir des données différentes. Il est donc indispensable de combiner plusieurs procédures de décision afin de traiter le plus efficacement possible chaque partie du système,
- Afin d'appeler le plus souvent possible les procédures de décision les plus efficaces, une abstraction de certains sous-systèmes peut être utile,
- Enfin, il est avantageux de travailler dans un cadre uniforme afin de minimiser les coûts de traduction et d'initialisation des structures de données.

I.4 Systèmes de contraintes en domaines continus

Cette section présente succinctement les systèmes de contraintes et les mécanismes de résolution associés, et introduit mes contributions à la résolution de contraintes en domaines continus. Les systèmes de contraintes en domaines finis sont détaillés section III.2.1 et leur utilisation en vérification formelle des programmes est présentée sections III.3 et III.4. Les travaux sur les contraintes en domaines continus sont présentés chapitre V.

I.4.1 Système de contraintes

Un système de contraintes [2, 21] est défini par un ensemble de *variables* avec des *domaines* associés et un ensemble de *contraintes* qui sont des relations entre les variables qui limitent les valeurs qu'elles peuvent prendre. Les domaines peuvent être *finis* c'est-à-dire un ensemble discret et fini de valeurs, ou bien *continus* c'est-à-dire un intervalle qui détermine un sous-ensemble des nombres réels.

La *résolution* d'un système de contraintes fait appel à deux étapes principales qui sont appliquées de façon itérative. Le *filtrage* consiste à réduire le domaine des variables en éliminant les valeurs qui ne satisfont pas les contraintes. Un opérateur de filtrage vérifie les propriétés de contractance (le nouveau domaine est inclus dans l'ancien), de correction (le nouveau domaine contient toutes les solutions), et de monotonie (préserve la relation d'inclusion). La *recherche* consiste à fixer la valeur de certaines variables en utilisant des heuristiques. Cela permet, lors de l'étape de filtrage suivante, de réduire le domaine des autres variables. Ces deux étapes sont itérées jusqu'à ce qu'une solution soit trouvée. Dans le cas fini, il s'agit tout simplement d'une instanciation des variables avec des valeurs de leur domaine qui satisfont les contraintes. Dans le cas continu, il s'agit d'un intervalle susceptible de contenir une solution réelle du système de contraintes. Quand le système de contraintes n'admet aucune solution, les étapes de filtrage/recherche réduisent un de ses domaines à l'ensemble vide ; on dit alors que le système est *inconsistant*.

En pratique, l'étape de filtrage utilise une *relaxation* du système de contraintes c'est-à-dire un ensemble de propriétés plus faibles que celles énoncées par le système de contraintes lui-même. Plus précisément, les *consistances partielles* déterminent des ensembles de valeurs qui contiennent toutes les solutions du système initial mais contiennent également des valeurs qui ne sont pas solution. Les consistances partielles peuvent être *locales* et ne considérer qu'une contrainte à la fois. Elles peuvent aussi prendre en compte un sous-ensemble de contraintes ou un sous-ensemble des domaines initiaux, comme c'est le cas par exemple en domaines continus où l'on considère seulement la borne minimale et maximale des intervalles. Les consistances partielles sont donc plus simples à vérifier, tout en permettant de filtrer le domaine des variables puisque les valeurs qui ne satisfont pas la consistance partielle ne satisfont pas a fortiori le système de contraintes initial.

Les atouts majeurs de la programmation par contraintes sont sa *simplicité d'utilisation* (l'utilisateur décrit simplement son problème), son *fort pouvoir d'expression* grâce à des contraintes spécifiques dédiées à diverses applications (planification, ordonnancement), *son efficacité* (des systèmes de contraintes en domaines finis de

plusieurs milliers de variables et de contraintes peuvent être résolus en quelques secondes), et le fait de manipuler dans un *cadre uniforme* des données de types différents (entiers, booléens, réels). Les problèmes majeurs en terme d'efficacité sont les problèmes de *localité*, quand les consistances locales sont insuffisantes pour filtrer le domaine des variables (alors qu'en ayant une vision plus globale du système les domaines pourraient être filtrés), les problèmes de *convergence lente*, quand l'itération répétée des étapes de filtrage/recherche ne permet d'éliminer qu'un seul élément du domaine à la fois (par exemple pour le système contenant les deux contraintes $x = 2y + 1$ et $x = 2z$ [28]) et enfin dans le cas continu, le nombre très important de flottants qui rend inutilisable en pratique des algorithmes polynomiaux quand la complexité dans le pire des cas est atteinte.

Dans mes travaux, je me suis intéressée à deux aspects. D'une part, l'utilisation des contraintes en domaines finis comme une procédure de décision pour la vérification des programmes. Pour décider de la formule F, il suffit de poser le système de contraintes correspondant à $\neg F$. Si ce système est inconsistant, alors P est valide, sinon, les solutions de $\neg F$ fournissent des contre-exemples à la validité de P. D'autre part, j'ai travaillé sur les consistances partielles dans le cas des domaines continus ; ceci est détaillé dans la sous-section suivante.

I.4.2 Contributions à la résolution de contraintes en domaines continus

L'étape de filtrage pour résoudre un système de contraintes en domaines continus fait appel à des consistances partielles qui sont d'une part locales à une contrainte et qui d'autre part manipulent uniquement les bornes des intervalles. Les deux familles de consistances les plus utilisées sont la "2B–consistance" et la "box–consistance"[5]. Soit une contrainte c et une variable x. De façon simplifiée, on peut dire que le domaine de la variable x est 2B–consistant s'il est le plus petit intervalle qui englobe les valeurs qui satisfont c quand la valeur de x est fixée à la borne supérieure (et inférieure) de son domaine. La box–consistance travaille elle sur l'extension optimale aux intervalles de la contrainte (voir section V.1 page 210 pour la définition de l'extension optimale aux intervalles d'une fonction réelle). Le domaine de la variable x est box–consistant s'il est le plus petit intervalle qui englobe les valeurs qui satisfont l'extension aux intervalles de c quand la valeur de x est fixée à l'intervalle réduit à la borne supérieure (et inférieure) de son domaine. De façon opérationnelle, la 2B–consistance nécessite une transformation du système en contraintes pour lesquelles on sait calculer les fonctions de projection qui assurent la 2B–consistance du domaine d'une variable. La box–consistance fait elle appel à l'extension aux intervalles de la méthode de Newton pour trouver le zéro le plus à gauche et le plus à droite de l'extension aux intervalles de la contrainte.

Notre première contribution est une comparaison fine entre ces deux familles de consistance. En particulier, l'originalité de ce travail de synthèse est d'avoir comparé le filtrage obtenu par ces consistances non seulement en fonction de leur définition, mais en tenant compte également de leur mise en œuvre pratique.

[5]Je parle ici de "famille" car la 2B–consistance (resp. la box–consistance) est la base d'une consistance d'ordre supérieur nommée "3B–consistance" (resp. "bound–consistance").

> Ce travail a été bien accueilli dans la communauté car la 2B–consistance était une méthode émergente prometteuse qu'il était donc intéressant de comparer à la box–consistance. Notons que ce travail de synthèse et de comparaison a été en partie induit par les questions que je me suis naturellement posées en découvrant la programmation par contraintes (ces travaux se situent au moment où j'ai effectué ma conversion thématique de la vérification formelle des processeurs vers la programmation par contraintes).

Notre deuxième contribution a été motivée par des applications pratiques en ingénierie mécanique où le concepteur connaît une solution possible, mais veut connaître le degré de liberté qu'il a sur les domaines des variables afin de réduire les coûts de production (e.g. une résistance avec une plus grande tolérance est moins coûteuse). Il s'agit donc de partir d'une solution et d'agrandir le domaine d'une variable en s'assurant que le nouveau domaine contient uniquement des solutions.

> Nous avons défini une nouvelle consistance, la "i–consistance" (pour consistance intérieure) qui définit une boîte qui contient uniquement des tuples solution, et avons proposé un algorithme pour étendre le domaine d'une variable en préservant la propriété de i–consistance. Le point fort de l'approche est l'efficacité de sa mise en œuvre grâce à la box–consistance.

Bibliographie

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager et P. H. Schmitt : The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] Krzystof R. Apt : *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] Peter J. Ashenden : *The VHDL Cookbook*. Public Domain, Dept. Computer Science, University of Adelaide, South Australia, first édition, juillet 1990.

[4] T. Ball, B. Cook, V. Levin et S. K. Rajamani : SLAM and static driver verifier : Technology transfer of formal methods inside microsoft. *Integrated Formal Methods (IFM)*, 2999, 2004.

[5] Kent Beck et Erich Gamma : Junit testing framework. `http://www.junit.org/`.

[6] Boris Beizer : *Software Testing Techniques*. International Thomson Computer Press, second édition, June 1990.

[7] A. Biere, A. Cimatti, E. M. Clarke et Y. Zhu : Symbolic model checking without bdds. *In TACAS*, volume 1579 de *LNCS*, pages 193–207. Springer-Verlag, 1999.

[8] Randal E. Bryant : Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, août 1986.

[9] Randal E. Bryant : Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), septembre 1992.

[10] Randal E. Bryant : Binary decision diagrams and beyond : Enabling technologies for formal verification. *In IEEE Internationl Conference on Computer-Aided Design*, pages 236–243, San Jose, CA (USA), 1995. IEEE Computer Society Press.

[11] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan et D.L. Dill : Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[12] J.R. Burch, E.M Clarke, K.L. McMillan, D.L. Dill et L. J. Hwang : Symbolic model checking : 10^{20} states and beyond. *In Fifth Annual Symposium*

on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 1990. IEEE Computer Society.

[13] L. Burdy, A. Requet et J. Lanet : Java applet correctness : a developer-oriented approach. *In Formal Methods (FME'03)*, volume 2805 de *LNCS*, pages 422–439. Springer Verlag, 2003.

[14] E.M. Clarke et E. Emerson : Design and synthesis of synchronization skeletons using branching time temporal logic. *In Logics of Programs*, volume 131 de *LNCS*. Springer-Verlag, 1981.

[15] E.M. Clarke, E. Emerson et A. Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):224–263, 1986.

[16] E.M. Clarke, O. Grumberg et D.E. Long : Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, September 1994.

[17] E.M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu et Helmut Veith : Counterexample-guided abstraction refinement for symbolic model checking. *Journal of ACM*, 50(5):752–794, 2003.

[18] E.M. Clarke, D. Kroening, N. Sharygina et K. Yorav : Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25:105–127, 2004.

[19] E.M. Clarke, D. Kroening, N. Sharygina et K. Yorav : SATABS : SAT-based predicate abstraction for ANSI-C. *In TACAS, Tool session*, volume 3440 de *LNCS*, pages 570–574. Springer-Verlag, 2005.

[20] Avra Cohn : The notion of proof in harware verification. *Journal of automating reasoning*, 5:127–139, 1989.

[21] Rina Dechter : *Constraint Processing*. Morgan Kaufmann publisher, 2003.

[22] David Déharbe : *Logic for Concurrency and Synchronisation*, volume 18 de *Trends in logic*, chapitre A tutorial introduction to model checking, pages 215–238. Kluwer Academic Publisher, 2003.

[23] Vijay D'Silva, Daniel Kroening et Georg Weissenbacher : A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[24] E. Emerson : Temporal and modal logic. *Handbook of Theoretical Computer Science*, B:995–1072, 1990.

[25] J.C. Filliâtre et C. Marché : The why/krakatoa/caduceus platform for deductive program verification. *In CAV*, volume 4590 de *LNCS*, pages 173–177. Springer-Verlag, 2007.

[26] Christoph Kern et Mark R. Greenstreet : Formal verification in hardware design : A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.

[27] James C. King : Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[28] Michel Leconte et Bruno Berstel : Domaines de congruence pour la programmation par contraintes. *In JFPC*, 2007.

[29] J.P. Queille et J. Sifakis : Specification and verification of concurrent systems in cesar. *In International Symposium on Programming*, volume 137 de *LNCS*. Springer-Verlag, 1982.

[30] I. Sommerville : *Software engineering*. Addison-Wesley Publishing Company, seventh édition, 2004.

[31] Stephen A. Ward et Robert H. Halstead : *Computation Structures*. MIT Press, 1989.

Chapitre II

Vérification formelle des processeurs

Ce chapitre présente mes travaux sur la vérification formelle des processeurs, travaux menés en collaboration avec Jacques Chazarain et Laurent Arditi, dont j'ai co-encadré la thèse.

Je décris d'abord la problématique de la vérification formelle des processeurs, telle qu'énoncée à l'époque de ces recherches (début des années 90). Je détaille ensuite nos principales contributions à la vérification formelle des processeurs et je conclus sur les points marquants des recherches actuelles.

Ce chapitre suppose une connaissance minimale de la structure des microprocesseurs ; le lecteur intéressé peut se référer au livre de S. Ward et Robert Halstead [51] qui détaille l'architecture d'un processeur depuis le niveau des portes logiques et le livre de W. Stallings [48] pour les évolutions récentes du domaine (cache, exécution dans le désordre, prédiction de branchement, ...), illustrées sur des processeurs réels.

II.1 Problématique et état de l'art

La vérification formelle de matériel est un domaine très vaste, incluant aussi bien la vérification d'une instruction assembleur, la vérification d'un protocole (e.g. vérifier qu'un contrôleur de bus va toujours satisfaire au plus une requête à la fois), ou la vérification d'une opération de calcul (e.g. multiplication en calcul flottant). Comme énoncé dans l'état de l'art de Christoph Kern et Mark R. Greenstreet [32], le problème de la vérification formelle de matériel peut se décomposer en deux grandes catégories, selon le type de spécification :

1. la spécification est une *propriété spécifique*, le plus souvent exprimée en *logique temporelle*. Il s'agit alors de montrer que tous les comportements possibles du système satisfont la propriété temporelle spécifiée,
2. la spécification est un *modèle de haut niveau* qui représente le comportement "correct" du système. Il s'agit alors de montrer que ce modèle est correctement implémenté par une réalisation d'un niveau d'abstraction plus concret.

Ces deux démarches peuvent être utilisées conjointement en vérifiant que le modèle de haut niveau vérifie certaines propriétés temporelles et en montrant ensuite que ce modèle de haut niveau est correctement implémenté. Le problème que nous avons considéré pour la vérification des processeurs entre dans la catégorie 2 ci-dessus. La vérification consiste à établir les relations entre le modèle de haut niveau nommé *spécification* et un niveau d'abstraction plus concret que l'on nomme *implémentation*.

II.1.1 Les niveaux d'abstraction

Les niveaux généralement considérés lors de la vérification d'un processeur sont les suivants, du plus abstrait au plus concret :

1. *niveau instruction assembleur* : il correspond à la vision du processeur qu'a le programmeur en assembleur.
2. *niveau séquence* : chaque instruction est exécutée par une suite de séquences déterminées par les échanges avec la mémoire. Les séquences principales sont

la lecture d'une instruction, son décodage, la lecture des opérandes, l'exécution et l'écriture du résultat. Ces séquences correspondent en fait aux *étages* des architectures pipeline, et peuvent donc être exécutées en parallèle pour des instructions différentes.

3. *niveau des micro-instructions* : une micro-instruction est une opération élémentaire qui peut être exécutée en un cycle horloge du processeur (e.g. addition de deux registres internes). Le terme de *micro-instruction* provient des architectures micro-programmées, où la partie contrôle du processeur lit successivement les micro-instructions dans la mémoire de micro-code pour envoyer les signaux correspondants au chemin de données (un bit de la micro-instruction est un signal du chemin de données). Notons que la notion de micro-programme existe encore dans les architectures actuelles, en particulier dans la famille des processeurs x86, pour des raisons de compatibilité ainsi que pour certaines instructions complexes[1].
4. *niveau transfert de registres* : à ce niveau on décrit la structure du processeur par un ensemble de blocs fonctionnels (unités de calcul et unités de stockage) connectés entre eux par des bus et des signaux. Chaque bloc est décrit soit par un ensemble de blocs plus simples, soit directement par son comportement.
5. *niveau portes* : chaque bloc est décrit par un réseau de portes logiques.
6. *niveau transistors* : chaque porte est décrite par un réseau de transistors.

La vérification formelle s'applique principalement aux niveaux les plus abstraits c'est-à-dire du niveau 1 au niveau 4. Pour les niveaux plus concrets, du niveau 4 au niveau 6, des outils de synthèse entièrement automatiques permettent de générer une implémentation dans le niveau concret à partir de sa spécification dans le niveau abstrait[2]. Notons que l'objet de ma thèse de doctorat a été la formalisation du *niveau séquence* et en particulier du parallélisme dans les architectures pipeline. La vérification établissait l'équivalence entre une instruction assembleur et une suite de micro-séquences par des techniques de réécriture.

Un niveau d'abstraction du processeur est classiquement décrit par son *état* : un ensemble de variables, et son *comportement* : un ensemble de *transitions d'état* [40, 27]. Par exemple au niveau assembleur, l'état est l'ensemble de tous les composants manipulables par un programme en assembleur et chaque instruction assembleur est décrite par une transition d'état, qui indique quelles variables ont été modifiées et comment.

II.1.2 La vérification

Pour vérifier un processeur, il faut montrer que chaque transition de la spécification est correctement réalisée par une séquence de transitions de l'implémentation.

[1] L'utilisation de micro-code permet de plus de corriger certaines erreurs dans l'architecture en fournissant un nouveau micro-code, voir les nombreux sites d'utilisateurs Linux ou Windows qui proposent des "patchs" de correction de micro-code, en particulier la correction d'un problème dans le TLB (Translation Lookaside Buffer du Intel Core2 Duo processeur [43]).

[2] Cette approche par synthèse est souvent complétée par des méthodes formelles d'équivalence d'automates car les concepteurs peuvent retoucher à la main l'implémentation qui a été synthétisée et ils veulent donc vérifier que cela n'a pas perturbé la correction du système.

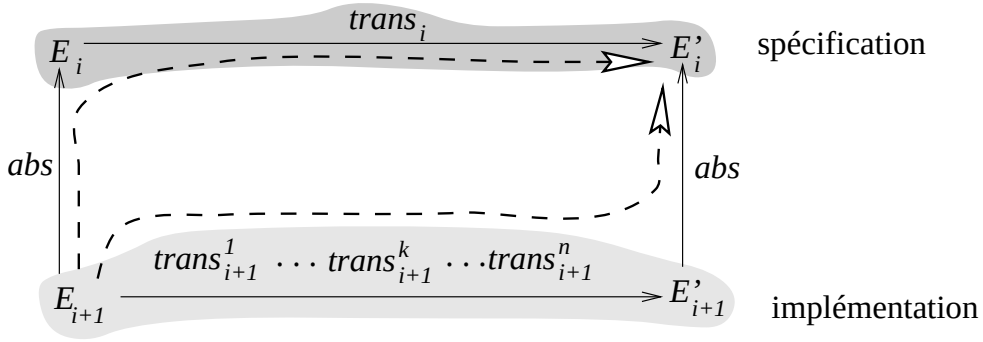


FIG. II.1 – Diagramme de preuve des instructions simples

Les états des deux niveaux sont mis en relation à travers des fonctions d'abstraction. Réaliser la preuve d'une instruction de la spécification consiste alors à vérifier que le diagramme de la figure II.1 commute.

Dans la figure II.1 le niveau i est celui de la spécification, le niveau $i+1$ celui de l'implémentation. Au niveau i, on a un état initial E_i et un état final E'_i. La transition $trans_i$ est telle que :

$$trans_i(E_i) = E'_i$$

Au niveau $i+1$, on a une séquence de transitions telle que :

$$trans^n_{i+1}(trans^{n-1}_{i+1}(\ldots(trans^1_{i+1}(E_{i+1}))\ldots)) = E'_{i+1}$$

La fonction d'abstraction est notée *abs*. Vérifier la commutativité du diagramme figure II.1 consiste à prouver que les deux transitions de E_{i+1} vers E'_i sont équivalentes, c'est-à-dire que

$$trans_i(abs(E_{i+1})) = abs(trans^n_{i+1}(trans^{n-1}_{i+1}(\ldots(trans^1_{i+1}(E_{i+1}))\ldots))) \qquad \text{(II.1)}$$

L'équation II.1 est implicitement universellement quantifiée : elle doit être vérifiée quel que soit E_{i+1} état valide c'est à dire atteignable après le reset. La grande majorité des travaux portant sur la vérification fonctionnelle des processeurs visent à vérifier des équations de cette forme.

II.1.3 État de l'art

À l'époque de nos travaux, c'est-à-dire au début des années 1990, les points notables de l'état de l'art sur la vérification formelle des processeurs étaient les suivants [26] :

- Les travaux précurseurs de Mike Gordon avaient montré comment *modéliser le matériel en logique d'ordre supérieur* en décrivant directement le matériel à vérifier dans l'assistant de preuve HOL [25, 24][3]. Les portes logiques (OR,

[3]Une présentation succincte de HOL4 est faite chapitre VI section IV.1 où je présente mes travaux sur la vérification des programmes basée sur la sémantique.

AND, XOR, ...) sont directement modélisées par les opérateurs booléens correspondants. Les connexions entre les portes sont exprimées par un quantificateur existentiel. Enfin, quand un modèle discret du temps est suffisant, le comportement temporel est modélisé par des fonctions d'ordre supérieur, un signal étant une fonction des entiers naturels (points d'observation ou tops d'horloge) vers les booléens.

- Ces principes de modélisation avaient été déclinés dans la logique d'autres démonstrateurs de théorèmes, comme par exemple Nqthm qui repose sur une logique du premier ordre sans quantificateur sur les fonctions récursives totales, avec égalité [10]. Ici le comportement temporel est modélisé par des fonctions récursives dont un paramètre est un entier qui représente le temps.
- En utilisant ces principes de modélisation avec un démonstrateur de théorèmes, plusieurs *cas particuliers* de processeurs avaient été prouvés. Par exemple, le processeur Viper avec le démonstrateur de théorèmes HOL [18], le processeur FM8501 avec le démonstrateur de théorèmes Nqthm [28] ainsi que l'AAMP5, un processeur réel assez complexe conçu pour des systèmes de contrôle aéronautique avec PVS (Prototype Verification System) [39, 47, 35][4]. Ces exemples précurseurs et convaincants avaient toutefois le défaut de nécessiter une *forte intervention humaine*, aussi bien pour la spécification que pour la preuve elle-même. De plus, il s'agissait de *cas particuliers*, et une partie seulement du travail de formalisation pouvait être réutilisé pour traiter un autre processeur.
- Une *méthodologie générale* basée sur la notion d'interprètes génériques avait été dégagée [53, 54], mais comme elle était implémentée dans le démonstrateur de théorèmes HOL elle n'était pas automatique. De plus, certaines notions faisaient défaut comme le comportement temporel des variables ou la représentation des entiers sous forme de vecteur de bits.
- Des *méthodes automatiques* existaient, mais elles portaient sur des descriptions de bas niveau et n'offraient pas d'interface de spécification [9, 14].

Notons que pour la vérification de matériel au sens large (i.e. cas où la spécification est donnée par une formule en logique temporelle), les points forts étaient les suivants :

- L'intérêt des BDDs et de leurs dérivés était un fait établi, et ils commençaient à être intégrés dans de nombreux outils de CAO à usage industriel,
- Grâce aux BDDs, le model checking en devenant *symbolique* avait permis de vérifier des circuits séquentiels de grande taille,
- Enfin, les travaux de normalisation des langages de spécification de matériel avaient permis de définir le langage VHDL qui devenait un point d'entrée incontournable des outils de vérification de matériel.

[4] PVS est un système intermédiaire entre HOL et Nqthm : il offre une logique d'ordre supérieur comme HOL mais il est plus automatisé que ce dernier.

II.2 Contributions

Dans le contexte décrit ci-dessus, nos contributions principales ont été les suivantes :

1. Nous avons proposé un *modèle de spécification générique*, orienté objet, qui permet de décrire aussi bien les différents niveaux d'abstraction que les preuves elles-mêmes (ce modèle été présenté à la conférence ECOOP'95 [5], et il est inclus comme article support section II.4.2). À partir de ce modèle, et grâce à l'étude de cas d'un processeur de taille significative décrit dans le langage de description de matériel VHDL [8], nous avons identifié un sous-ensemble de VHDL que nous sommes capables de traiter (voir EURODAC'95 [6]).
2. Notre méthodologie de preuve est *automatique* dans la plupart des cas. Elle intègre des techniques de vérification *hybrides*. Un moteur *d'exécution symbolique* et un système de *calcul formel* spécifique permettent de calculer de façon efficace les transitions d'état [15]. Une représentation compacte des *vecteurs de bits* grâce aux **BMD* (Multiplicative Binary Moment Diagrams) permet de retarder l'explosion combinatoire des expressions obtenues dans le cas d'instructions implémentées par une boucle de longueur fixe [3]. Enfin, l'assistant de preuve *Coq* est utilisé pour effectuer la preuve de façon *inductive* quand il s'agit de vérifier une instruction assembleur contenant une boucle de longueur inconnue.
3. Ce modèle et cette méthodologie ont été implémentés et appliqués à la preuve de *nombreux processeurs, de taille significative*. En particulier, nous avons vérifié de façon automatique des processeurs qui avaient été vérifiés de façon manuelle, et nous avons vérifié un processeur réel, le MTI, conçu par le CNET à Grenoble. De plus, nous avons vérifié des *instructions de boucle*, ce qui n'avait pas été fait jusqu'alors.

Le point 1 répond à un premier objectif qui est de fournir un *cadre général* qui permette la vérification de plusieurs processeurs. Il s'agit aussi d'offrir à l'utilisateur une *interface de description* des processeurs aussi proche que possible de celles qu'il utilise pour les autres méthodes de CAO, comme la simulation. Le point 2 a pour objectif d'assurer *la rapidité de la vérification* et de *minimiser l'intervention humaine* dans la vérification. Les méthodes efficaces (et incomplètes) sont utilisées en premier. L'intervention humaine est différée au maximum, elle n'est sollicitée qu'en dernier recours, pour donner par exemple un invariant de boucle. Notons que ces objectifs sont aussi ceux que j'ai visés dans mes travaux sur la vérification de programmes (voir chapitre III).

Dans ces travaux, la méthodologie générale a été définie par Jacques Chazarain et moi-même. Laurent Arditi l'a mise en œuvre et a apporté une forte contribution avec l'intégration des *BMD [3] et l'utilisation de l'assistant de preuve *Coq* [4]. L'article de la revue TSI [7] a été inclus comme article support en section II.4.1 page 40 car il présente une vue d'ensemble de ces travaux. J'en détaille ci-dessous les points forts.

II.2.1 Méthodologie de spécification

Les interprètes Le cœur de notre méthodologie de spécification est la notion d'*interprète*, notion communément adoptée en vérification formelle [2, 30, 52, 27, 14].

Un interprète est modélisé à un niveau i par un triplet $(state, select, instructions)$ tel que :

- *state* : représente l'ensemble des états du processeurs. Chaque état est le n-uplet des valeurs des variables utilisables par les programmes du niveau i.
- *select* : est une fonction de sélection d'instruction. Son domaine est *state* et son co-domaine est un ensemble de clefs *keys*.
- *instructions* : un ensemble d'instructions modélisé par une fonction de *keys* dans l'ensemble des transitions d'état.

Un tel interprète est générique car il est réutilisable pour spécifier différents processeurs et s'applique à différents niveaux d'abstraction [52].

Exemple II.1 (État au niveau instructions assembleur) Pour le niveau *instructions assembleur*, l'état est l'ensemble des registres et mémoires accessibles au programmeur. La fonction de sélection associe au code opération qui est contenu dans le registre d'instruction IR (ce registre est une variable d'état du niveau instruction assembleur) un nom d'instruction assembleur (qui inclut le mode d'adressage). Pour chaque instruction, la transition d'états associée correspond à la description que l'on peut trouver dans un manuel de programmation assembleur. Par exemple, la transition d'états associée à l'instruction d'addition en mode "direct registre" modifie le registre compteur programme PC et le banc de registre REG de la façon suivante :

$PC = PC + 2$

$REG[IR[0:2]] = REG[IR[0:2]] + REG[IR[3:6]]$

On a supposé ici que la description du jeu d'instructions assembleur stipule que l'instruction est codée sur deux octets et que $IR[0:2]$ est le code du registre destination et $IR[3:6]$ le code du registre source. ♯

Les preuves Pour assurer la généricité de la preuve, nous avons étendu le concept d'interprète générique. Nous décrivons les vérifications à effectuer entre deux niveaux de spécification par une entité désignant deux interprètes et définissant les fonctions d'abstraction de l'implémentation vers la spécification :

- *abstraction* : définit l'abstraction de l'état. Le plus souvent, elle est unique car il s'agit d'une projection : l'état de la spécification est un sous-ensemble de l'état de l'implémentation.
- *sync* : définit l'abstraction temporelle. C'est un prédicat vrai seulement quand les deux niveaux sont synchronisés c'est-à-dire quand l'état de l'implémentation correspond au début d'une transition au niveau abstrait. Il est le plus souvent défini comme un test sur la valeur du compteur programme de l'implémentation.

Notons que de tels interprètes modélisent trois types d'abstraction parmi les quatre qui ont été identifiés comme pertinents pour la vérification de matériel par Tom Melham dans [34]. *L'abstraction temporelle* qui relie les échelles de temps (e.g. pour les processeurs, une unité de temps est l'exécution d'une instruction assembleur

pour la spécification alors que l'unité de temps de l'implémentation est un top d'horloge). *L'abstraction structurelle* qui supprime les détails de la structure interne de l'implémentation. Il s'agit d'une vision "boîte noire" : les détails de l'implémentation sous forme de portes logiques sont cachés quand on effectue la preuve entre le niveau assembleur et transfert de registres. Enfin, *l'abstraction des données* qui relie le format des données de l'implémentation à celui de la spécification (e.g. vecteurs de bits pour l'implémentation et entiers pour la spécification). Cette dernière abstraction est effectuée de façon concrète par des opérations de conversion des vecteurs de bits vers les entiers.

L'implémentation orientée objet Nous avons choisi une approche objet pour implémenter l'environnement de spécification, aussi bien des processeurs que des preuves [5] (voir article support section II.4.2 page 66). À l'époque de nos travaux, une telle approche était rare dans le cadre de la conception des circuits, mais un groupe de travail visait à intégrer cette notion dans le langage VHDL [36, 42, 1]. De nos jours, l'approche objet n'a toujours pas été implémentée dans VHDL, mais cette notion a été intégrée dans SystemVerilog, en particulier pour la création de jeux de test.

La hiérarchie de classes a été implémentée dans le langage fonctionnel orienté objet STk [23], qui est un interprète Scheme associé à un système objet très proche de CLOS [49]. Pour aider les concepteurs qui ne sont pas familiers avec Scheme, nous avons conçu un langage simple pour décrire les processeurs dans un style facilitant la preuve. Sa syntaxe est proche du standard VHDL et nous avons montré dans [6] la faisabilité de la traduction d'un sous-ensemble de VHDL vers ce langage. À partir de cette forme textuelle, un traducteur génère automatiquement la représentation interne Scheme.

II.2.2 Méthodologie de vérification

Si les précédents travaux de vérification formelle des processeurs consistaient à vérifier la commutativité de diagrammes similaires à celui présenté figure II.1, notre étude considère un cadre plus général où la transition $trans_i$ est répétée p fois et une sous-séquence des transitions de l'implémentation est répétée q fois, comme présenté dans la figure II.2. Pour vérifier qu'un tel diagramme commute, nous avons défini une méthodologie de vérification hybride qui inclut :

1. l'exécution symbolique avec simplification,
2. l'exécution symbolique en utilisant la structure de données des *BMDs,
3. la preuve inductive avec l'assistant de preuve *Coq*.

Ces méthodes sont essayées en séquence et en fonction des valeurs du couple (p, q) du diagramme de la figure II.2. L'exécution symbolique avec simplification est entièrement automatique et très efficace, et s'applique dans de nombreux cas. L'exécution symbolique avec utilisation de la structure de données des *BMDs est automatique mais plus coûteuse, et est utilisée pour retarder l'explosion combinatoire quand le nombre d'itérations est connu dans le cas ($p = 0$ et $q \neq 0$). Enfin la preuve inductive n'est pas automatique, et est utilisée en dernier recours dans le cas

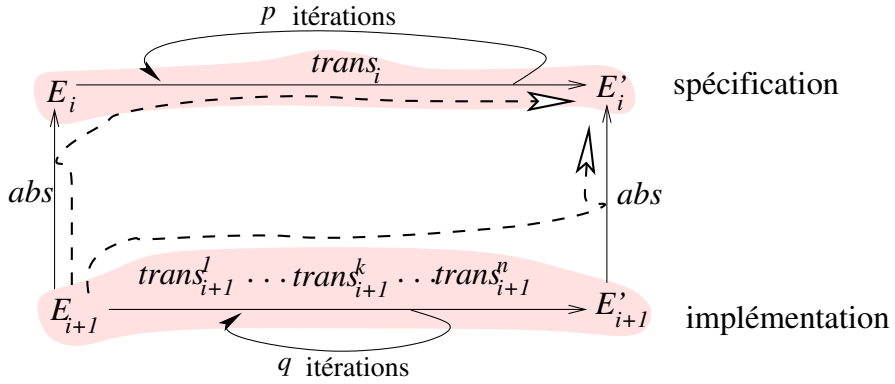


FIG. II.2 – Diagramme de preuve des instructions complexes

($p \neq 0$ et $q \neq 0$) ou dans le cas ($p = 0$ et $q \neq 0$) quand le nombre d'itérations dépend des données. Plus précisément :

1. Quand $p = q = 0$ il s'agit du cas de la figure II.1. Cela représente la très grande majorité des instructions des processeurs. L'exécution symbolique et la simplification sont suffisantes pour vérifier de telles instructions.

2. Quand $p = 0$ et $q \neq 0$, nous avons à traiter des *instructions de boucle*. Cela signifie que l'architecture du processeur ne permet pas de réaliser directement l'opération de $trans_i$ mais qu'une itération d'opérations plus simples au niveau concret est équivalente. C'est par exemple le cas des instructions arithmétiques complexes comme la multiplication ou la division pour des processeurs qui ne contiennent pas de circuits combinatoires réalisant ces opérations. Le micro-programme de ces instructions comporte alors une boucle. Dans le cas où le nombre d'itérations q est fixé par la taille des données, comme pour la multiplication par additions/décalages où q est égal à la taille des vecteurs de bits, l'exécution symbolique est possible. Cependant, la composition d'opérations élémentaires peut amener rapidement une explosion combinatoire dans le calcul des expressions. La représentation efficace des *BMDs permet de retarder cette explosion.

3. Quand $p \neq 0$ et $q \neq 0$, l'instruction de boucle du niveau abstrait est aussi implémentée par une boucle au niveau concret. Ceci concerne surtout les instructions de décalage et de manipulation de blocs de mémoire. Par exemple l'instruction *rep stosb* de la famille x86 [17] remplit une partie de la mémoire avec une valeur constante. Cette valeur, l'adresse de début du bloc et surtout la taille du bloc sont des valeurs de registres utilisateurs. Le nombre d'itérations est inconnu et une preuve inductive est effectuée avec l'assistant de preuve *Coq*.

4. Le cas où $p \neq 0$ et $q = 0$ représente une mauvaise spécification car si une instruction peut être décrite au niveau concret sans utiliser de boucle, alors il n'est pas nécessaire d'utiliser une boucle à un niveau plus abstrait.

nom	usage	fonction
`concat`	`concat`$(V_1, V_2, \ldots, V_n)$	concaténation
`field`	`field`(V, i, j) ou V`[`i `:`j`]`	sélection d'un champ de bits
`sum`	`sum`(V_1, V_2)	somme d'un additionneur
`carry`	`carry`(V_1, V_2)	retenue sortante d'un additionneur
`add`	`add`(V_1, V_2) équivalent à `concat(sum`(V_1, V_2)`,carry`(V_1, V_2)`)`	addition
`sub`	`sub`(V_1, V_2)	soustraction
`incr`	`incr(V)` équivalent à `sum(V,"1")`	incrémentation
`and`, `or`, `xor`	`and`$(V_1, V_2, \ldots, V_n)$	opérations logiques bit-à-bit
`not`	`not`(V)	négation logique
`mux`	`mux`(C, V_1, V_2) équivalent à `if` C `then` V_1 `else` V_2	sortie d'un multiplexeur
`BV_nat`	`BV_nat`(V) équivalent à $\$V$	conversion vers les entiers naturels
`exts`	`exts`(V, n)	extension à n bits
`length`	`length`(V)	longueur d'un vecteur de bits
`lsb`	`lsb`(V) équivalent à V`[0 :0]`	bit de poids le plus faible
`msbs`	`msbs`(V) équivalent à V`[1 :length(`V`)-1]`	bits de poids forts
`BV_null`	`BV_null`(n)	vecteur formé de n bits nuls
`eq`	`eq`(V_1, V_2) ou $V_1 = V_2$	égalité

TAB. II.1 – L'algèbre des vecteurs de bits

II.2.3 Les outils pour la vérification

Cette sous-section présente les outils de notre méthode de vérification hybride : le système de calcul formel, les *BMDs et l'assistant de preuve *Coq*.

Un moteur d'exécution symbolique et un système de calcul formel spécifique permettent de calculer l'état à un niveau donné, après exécution d'une séquence de transitions [15]. Ce système de calcul formel intègre une algèbre sur les entiers, les vecteurs de bits et les tableaux de vecteurs de bits, ainsi que les fonctions d'abstraction des données entre les entiers naturels et les vecteurs de bits (à titre indicatif, la table II.1 présente les opérations sur les vecteurs de bits que nous considérons). Le système de calcul formel est composé d'un noyau de simplification et d'un ensemble de définitions d'opérateurs : l'évaluation d'une expression est réalisée en profondeur d'abord et si l'opérateur de tête est connu, la fonction de simplification associée est appelée. L'exécution symbolique et le calcul formel ont été suffisants dans tous les exemples que nous avons traités en l'absence de boucle, c'est-à-dire quand $p = q = 0$.

Les *BMDs offrent une représentation symbolique compacte des vecteurs de bits. Comme les BDDs[5], les *BMDs [12] sont une représentation canonique des fonctions sur des variables booléennes par des graphes acycliques[6]. Les *BMDs fournissent une représentation canonique des fonctions de $\{0,1\}^n$ dans $\mathbb{Z}$, tandis que les BDDs fournissent une représentation canonique des fonctions de $\{0,1\}^n$ dans $\{0,1\}$. L'atout

[5]Les BDDs sont présentés annexe VII.1.3 page 269.

[6]Comme pour les BDDs, pour que la représentation soit canonique il faut utiliser un ordre sur les variables et partager les sous-graphes.

des *BMDs est que la représentation non signée des mots et la représentation de l'addition ont une taille qui croît linéairement avec le nombre de bits. C'est exactement ce que nous demandons pour traiter des instructions complexes qui implémentent des opérations arithmétiques sur les vecteurs de bits.

Soit f une fonction de $\{0,1\}^n$ dans $\mathbb{Z}$. Les *BMDs sont basés sur l'expansion de Reed-Muller de f par rapport à une variable booléenne x. Nous notons $f_{\overline{x}}$ la restriction de f quand $x = 0$ et f_x sa restriction quand $x = 1$. L'expansion de Shannon de f par rapport à x donne tout d'abord[7]

$$\begin{aligned} f &= (1-x)\cdot f_{\overline{x}} + x \cdot f_x \\ &= f_{\overline{x}} + x \cdot (f_x - f_{\overline{x}}) \end{aligned}$$

f est ainsi décomposée en deux moments :

$$f = f_{\overline{x}} + x \cdot f_{\dot{x}} \qquad \text{où} \qquad f_{\dot{x}} = f_x - f_{\overline{x}}$$

$f_{\overline{x}}$ est le *moment constant*. Il représente la valeur de la fonction quand $x = 0$. L'autre moment, $f_{\dot{x}}$, est le *moment linéaire*. Il représente la variation de f quand x passe de 0 à 1.

Un *BMD est donc un arbre ayant un identificateur de variable comme racine et deux *BMDs comme fils qui sont les moments constant et linéaire. Les *BMDs ont des valeurs numériques associées aux feuilles mais aussi aux branches. Ces valeurs sont multipliées au cours du parcours d'un arbre d'où le nom de *Multiplicative* Binary Moment Diagram (*BMD). Comme pour les autres représentations symboliques équivalentes, les tailles des *BMDs sont réduites en partageant des sous-arbres communs. Bryant et Chen ont proposé des fonctions de synthèse en appliquant l'addition, la multiplication ou l'exponentiation à des *BMDs tout en conservant une forme canonique [11].

Exemple II.2 (Représentation avec les *BMDs) La figure II.3 montre les représentations de quelques opérations avec des *BMDs. Par exemple, on peut voir en **(a)** que la valeur non signée du vecteur composé des bits x_0 (poids faible), x_1 et x_2 est $4 * x_2 + (2 * x_1 + (1 * x_0 + 0)) = 4 * x_2 + 2 * x_1 + x_0$. On peut voir en **(c)** le produit des vecteurs X (x_0, x_1, x_2) et Y (y_0, y_1, y_2). La fonction ainsi représentée vaut $4*x_2*(4*y_2+2*y_1+y_0)+2*x_1*(4*y_2+2*y_1+y_0)+x_0*(4*y_2+2*y_1+y_0) = (4 * x_2 + 2 * x_1 + x_0) * (4 * y_2 + 2 * y_1 + y_0)$ ce qui est bien la valeur de $X * Y$.

Nous avons étendu les *BMDs dans le but de fournir tous les opérateurs de l'algèbre sur les vecteurs de bits présentée table II.1. Nous utilisons les *BMDs exactement comme s'ils représentaient des vecteurs de bits, et non pas des entiers, en tenant compte de la taille des vecteurs. Alors que les *BMDs représentent des fonctions de $\{0,1\}^n$ vers $\mathbb{Z}$, nous avons défini les $\overrightarrow{\text{*BMD}}_s$ qui représentent les fonctions de $\{0,1\}^n$ vers les entiers de $[0 : 2^s[$. Des détails sur les $\overrightarrow{\text{*BMD}}_s$ peuvent être trouvés dans l'article support de la section II.4.1, dans [3], ainsi que dans la thèse de Laurent Arditi [4].

[7] $+, -, \cdot$ représentent l'addition, la soustraction et la multiplication pour entiers.

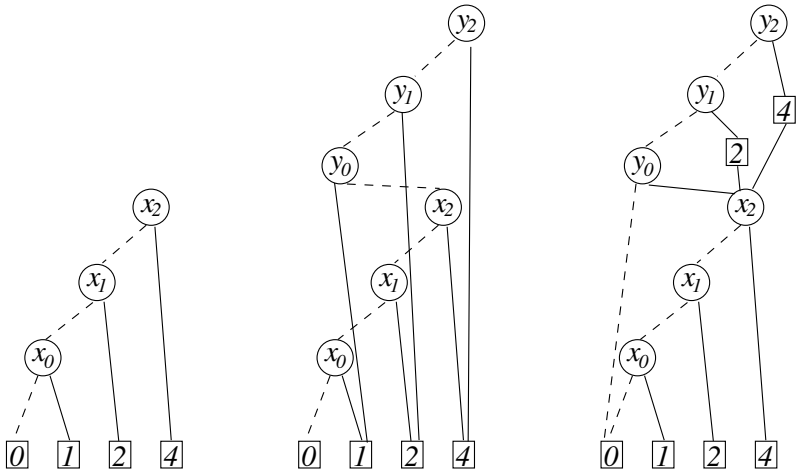


FIG. II.3 – Représentation par des *BMD d'un mot non signé **(a)**, de l'addition **(b)** et de la multiplication **(c)**. Les valeurs des branches égales à 1 ne sont pas indiquées, les lignes pointillées sont les moments constants, les lignes pleines les moments linéaires.

Les *BMDs ont permis de résoudre le cas où ($p = 0$ et $q \neq 0$) et où le nombre d'itérations est connu dans la figure II.2. En particulier, une instruction de multiplication par addition/décalage a été vérifiée automatiquement en utilisant l'exécution symbolique et les *BMDs, et ce pour des vecteurs de bits de taille 128[8]. Cette vérification n'avait pas été possible en utilisant seulement notre système de calcul formel. Cependant, les *BMDs ne permettent pas de vérifier la division, qui est pourtant implémentée par un micro-programme proche de celui de la multiplication. C'est une conséquence du fait que le micro-programme contient un test comparatif $<$ entre deux *BMDs et que la vérification d'un tel test est exponentielle avec les $*BMD$ (alors que cela est linéaire avec les BDDs) [4].

Preuves inductives avec l'assistant de preuve *Coq* Dans notre méthodologie de preuve, l'induction est utilisée en dernier recours car elle nécessite l'intervention humaine. Notre choix s'est donc porté sur un *assistant de preuves* plutôt qu'un démonstrateur plus automatique comme Nqthm. De plus, nous avons besoin d'un système basé sur une *logique d'ordre supérieur*, pour pouvoir modéliser des circuits séquentiels, où les composants sont représentés par des fonctions sur le temps. À l'époque de nos recherches, les deux assistants de preuve satisfaisant les critères ci-dessus étaient HOL et Coq. De nombreux travaux sur la vérification de matériel avaient déjà été réalisés avec HOL, alors que l'utilisation de Coq pour ce problème en était à son début, avec la vérification de circuits combinatoires ou séquentiels simples décrits au niveau des entiers [22, 41]. Cependant, Coq étant développé à l'INRIA, en partie par des équipes ayant des liens étroits avec notre laboratoire,

[8]Dans ce cas, la taille des *BMDs croît de façon linéaire en fonction de la taille des vecteurs de bits

notre choix s'est naturellement porté sur *Coq*.

Coq est un assistant de preuve pour le λ-calcul d'ordre supérieur. Ses bases théoriques sont le Calcul des Constructions [19] étendu avec des définitions inductives [20]. Le développement d'une preuve correspond à la construction d'un λ-terme ; cette caractéristique est exploitée par la possibilité de synthétiser un programme à partir de la preuve de sa spécification. Le lecteur est invité à se référer à [21] pour une description détaillée de ce système.

Nous avons implémenté en *Coq* tous les types de données de l'algèbre de la figure II.1. Les booléens et les entiers naturels sont modélisés par les théories standard et le type des vecteurs de bits a été défini par des listes polymorphes, de même que les mémoires. L'abstraction de données, réalisée par une conversion des vecteurs de bits vers les entiers naturels, a été définie de façon inductive. Il est important de noter que les théorèmes que nous avons prouvés sur les vecteurs de bits correspondent aux règles de simplification de notre système de calcul formel. Le développement de la théorie des vecteurs de bits en *Coq* est donc une validation partielle du noyau de simplification de notre système.

De façon voisine à celle adoptée avec Nqthm dans [29], les fonctions de transition d'états sont modélisées par un ensemble de fonctions qui donnent la valeur des composants à l'instant t en fonction de leurs valeurs à l'instant $t-1$. Les variables modifiées à l'intérieur des boucles sont représentées par des fonctions récursives sur le nombre de passages dans la boucle. La preuve d'une instruction de boucle consiste à montrer qu'un prédicat sur les valeurs des composants et/ou des sorties, est vrai quand une condition est vérifiée. La preuve demande souvent une généralisation, particulièrement quand le prédicat est une fonction primitive comme la multiplication. L'utilisateur doit trouver un invariant de boucle et le prouver, le théorème final n'est alors qu'une spécialisation de cet invariant. Les détails sur l'utilisation de *Coq* peuvent être trouvés dans la thèse de Laurent Arditi [4].

II.3 Conclusion

Nous avons proposé une approche pour la vérification des processeurs qui est automatique dans la plupart des cas et qui permet de vérifier aisément une large classe de processeurs, dont les spécifications peuvent être traduites facilement depuis un sous-ensemble de VHDL que nous avons identifié. Cette méthodologie a été implémentée et appliquée à plusieurs processeurs de taille significative qui avaient été vérifiés "à la main" avec un assistant de preuve dans les travaux précédents. De plus, nous avons traité des instructions de boucle ce qui n'avait pas été abordé à l'époque de ces travaux.

Ayant changé de thème de recherche en 1997, je n'ai pas suivi de près l'évolution des méthodes de vérification formelle du matériel[9]. Je ne fais donc pas ici un état de l'art des recherches actuelles mais je me contente de noter quelques points pertinents par rapport à l'approche que nous avons proposée.

Tout d'abord, les différentes approches de vérification formelle (manuelles ou

[9] J'ai par contre suivi de près l'évolution de l'architecture des unités centrales puisque j'ai été responsable du cours d'architecture des ordinateurs pendant une dizaine d'années.

automatiques) ont été appliquées sur les nouveaux types d'architectures comme les architectures pipelines, super-scalaires et VLIW (Very Long Instruction Word). Dans ces architectures, la difficulté est de prendre en compte le parallélisme d'exécution des instructions dans la formalisation de la preuve. Par exemple, dans une architecture pipeline, l'état courant est à la fois l'état final d'une instruction et un état intermédiaire pour les autres instructions dans le pipeline. Les travaux précurseurs de Burch et Dill [14] ont introduit un nouveau diagramme de commutativité entre le niveau instruction assembleur et transfert de registres qui prend en compte le pipeline : les états sont mis en relation seulement quand le pipeline a été vidé. Un exemple d'approche manuelle est donné dans [44] où le diagramme de commutativité a été amélioré pour vérifier avec $ACL2$ [31] (le successeur de $Nqthm$) un processeur avec exécution dans le désordre. Il s'agit du $FM9801$, successeur du $FM8501$ qui a été l'un des premiers processeurs formellement vérifié avec $Nqthm$ par W. Hunt. La preuve a été menée mécaniquement dans $ACL2$ en deux étapes : montrer un certain nombre d'invariants à savoir des propriétés sur le comportement des instructions en cours d'exécution parallèle (e.g. il n'y a pas de conflit structurel dans le pipeline, les instructions sont dispatchées dans le bon ordre) puis effectuer la preuve entre le niveau instruction assembleur et transfert de registres. Enfin, un exemple de vérification complètement automatique est donnée dans [45]. Le système est décrit dans une logique du premier ordre avec fonctions non-interprétées, égalité et lambda expressions, et la vérification inclut un simulateur symbolique et des procédures de décision spécifiques. L'approche a permis la vérification d'invariants pour l'exécution dans le désordre, et en particulier pour l'algorithme de Tomasulo.

Un autre point concerne l'usage de la structure des $*BMD$, qui est une des originalités de notre approche. Cette structure venait d'être définie par R. Bryant. Bien qu'efficace pour représenter certaines opérations arithmétiques comme $+, *, exp$, cette structure n'a pas été aussi largement utilisée que celle des BDD. En effet, elle ne permet pas de trouver le zéro d'une fonction efficacement (la structure n'est pas "inversible"), ce qui est nécessaire pour vérifier certaines propriétés. R. Bryant et Yirng-An Chen ont donc proposé dans [13] d'utiliser les $*BMD$ conjointement aux BDD dans le cadre d'une vérification modulaire par niveaux d'abstraction, pour les circuits arithmétiques. L'idée est de profiter de l'efficacité des $*BMD$ pour les niveaux abstraits (i.e. pour représenter directement des entiers ou des flottants), et de les combiner avec des BDD pour les niveaux plus concrets ou quand l'inversion est nécessaire. Notons que les mêmes auteurs présentent dans [50] une comparaison entre l'utilisation des $BDDs$ et des solveurs SAT pour la vérification de processeurs super-scalaires et VLIW. Plus précisément, ils vérifient la correction de l'architecture super-scalaire/VLIW par rapport à une version non pipeline. ils utilisent la théorie des fonctions non-interprétées. Cette comparaison a montré que les solveurs SAT, et en particulier le solveur $Chaff$ sont plus efficaces que les $BDDs$. Les $BDDs$ restent néanmoins une structure de données utilisée dans bon nombre d'outils de CAO.

Enfin, l'association d'assistants de preuve et d'outils automatiques est largement d'actualité. En particulier, les démonstrateurs comme HOL4 ou ACL2 intègrent des procédures de décision (e.g. solveur SAT) qui peuvent être appelées directement depuis le démonstrateur. Nous reviendrons là-dessus dans nos perspectives de recherche chapitre VI.

Pour terminer cette conclusion et introduire le chapitre suivant, je présente trois travaux qui sont au confluent de la vérification de matériel et de logiciel et qui ont de surcroît été menés par des chercheurs de renom dans la communauté de la vérification de matériel.

Eric Whitman Smith et David Dill[10] ont présenté dans [46] un cas d'étude concernant la vérification d'un algorithme de cryptographie (block ciphers). L'implémentation est un programme Java et la spécification est soit un autre programme Java soit une spécification formelle écrite dans le langage logique du démonstrateur de théorèmes ACL2. L'idée est de traduire aussi bien la spécification que l'implémentation (en fait le "byte code" Java qui est exécuté symboliquement) sous forme de termes qui sont simplifiés en utilisant un système ad-hoc basé sur les règles de simplification d'ACL2 puis prouvés égaux après traduction sous forme de vecteurs de bits par un solveur SAT. Il s'agit donc d'une approche très similaire au bounded model checking des programmes.

Lors de mon séjour à Cambridge dans l'équipe du professeur Mike Gordon, j'ai pu aussi constater une orientation de leurs travaux vers un aspect plus proche du logiciel. En effet, Magnus Myreen, Mike Gordon et Konrad Slind ont proposé une méthode de dé-compilation de programmes assembleur sous la forme de fonctions récursives définies dans HOL4 [38]. Le processus est automatique et générique car il s'applique à plusieurs jeux d'instruction (ARM, x86, PowerPC). L'idée est d'automatiser la traduction du jeu d'instructions assembleur d'un processeur afin de le mettre en relation avec un niveau d'abstraction plus concret. En particulier, l'architecture du ARM 5 a été spécifiée en totalité dans HOL4 et prouvée (manuellement) correcte par rapport à sa description au niveau transfert de registres. D'autre part, ces travaux ont permis de produire une implémentation du cœur du langage LISP 1.5 qui a été vérifiée formellement et automatiquement avec HOL4 [9].

Enfin, Edmund Clarke, Daniel Kroening et Karen Yorav ont présenté dans [33] une méthode de bounded model checking pour vérifier le comportement de circuits décrits en Verilog dont la spécification est donnée en ANSI-C. Le principe est de déplier le circuit et la spécification et de traduire la formule à prouver en une formule booléenne qui est ensuite résolue avec un solveur SAT. En particulier, ils ont vérifié l'architecture du processeur DLX, architecture RISC définie à Berkeley, et qui sert de support au livre d'architecture de John L. Hennessy et David A. Patterson "Computer Architecture, A Quantitative Approach"[11]. Ces travaux sont précurseurs des travaux sur la vérification de propriétés de sûreté et d'assertions de programmes C implémentés dans l'outil CBMC que j'ai comparé de façon expérimentale à notre outil de vérification de programmes par programmation par contraintes [13] (voir chapitre III section III.4.5 119).

[10] David Dill est connu pour ses travaux sur la vérification des processeurs ; il a accueilli Laurent Arditi qui a effectué un séjour post-doctoral dans son équipe.

[11] Ce livre est un des supports de mon cours d'architecture des ordinateurs.

II.4 Articles supports

II.4.1 Intégration de techniques coopératives pour la vérification formelle des processeurs

Laurent Arditi, Hélène Collavizza. Intégration de techniques coopératives pour la vérification formelle des processeurs. Technique et Science Informatiques, vol 16(6), June 1997.

Cet article est une synthèse de nos travaux. Il présente l'approche générale et donne des résultats expérimentaux.

Intégration de techniques coopératives pour la vérification formelle des processeurs

Laurent Arditi et Hélène Collavizza

Université de Nice – Sophia Antipolis
Laboratoire I3S, CNRS-URA 1376
650, Route des Colles. B.P. 145
06903 Sophia Antipolis Cédex. France
email: arditi@unice.fr, helen@essi.fr

RÉSUMÉ. *Nous proposons un environnement de spécification et vérification formelle des processeurs. Ce système intègre différentes techniques de preuve : simplification, représentation symbolique compacte sous forme de Diagrammes de Moments Binaires, preuve inductive avec* Coq*. Ces techniques sont utilisées de façon progressive afin de minimiser l'intervention de l'utilisateur. Notre système a permis de prouver efficacement plusieurs processeurs et des instructions complexes qui n'avaient jamais été vérifiées.*

ABSTRACT. *We propose a framework for the specification and formal verification of processors. This system integrates different proof techniques: simplification, a compact symbolic representation (Binary Moment Diagram), inductive proof with* Coq*. These techniques are used in a progressive way in order to reduce user intervention. Our system has been efficiently used to verify several processors and complex instructions.*

MOTS-CLÉS : *Vérification formelle, spécification, preuve, microprocesseur, exécution symbolique, simplification, Diagrammes de Moments Binaires,* Coq

KEY WORDS : *Formal verification, specification, proof, microprocessor, symbolic execution, simplification, Binary Moment Diagrams,* Coq

1. Introduction

1.1. *Le problème*

La complexité actuelle des microprocesseurs rend indispensable leur validation en cours de conception, avant de les graver définitivement sur silicium. La conception assistée par ordinateur (CAO) est incontournable pour concevoir des circuits ayant plusieurs milliers de transistors. Les outils de CAO usuels permettent de faire le schéma, le placement/routage des composants et la simulation du comportement des circuits.

Toutefois, la technique de validation largement implantée dans l'industrie est la simulation qui ne permet pas une vérification exhaustive, puisque les circuits sont simulés seulement pour certaines valeurs de leurs entrées et variables internes. De plus, les techniques actuelles de CAO sont plutôt dédiées aux composants électroniques et ne permettent pas de vérifier les visions abstraites des circuits.

L'application des méthodes de vérification formelle à la validation des circuits digitaux vise à combler ces deux lacunes. D'une part, elles fournissent des modèles suffisament abstraits. D'autre part, les preuves portent sur des formules universellement quantifiées et sont donc exhaustives. Plus précisément, vérifier formellement un circuit consiste à :

1. définir un modèle formel de la spécification du circuit ;
2. définir un modèle formel de l'implémentation du circuit, et
3. établir la preuve que pour toute valeur des entrées et des variables d'état, le modèle de l'implémentation implique celui de la spécification.

La vérification formelle à bas niveau commence à être utilisée en milieu industriel. Dans cet article, nous considérons un niveau abstrait, le jeu d'instructions des processeurs, pour lequel il n'existe pas encore d'outil de CAO. Il s'agit dans ce cas de montrer qu'un jeu d'instructions assembleur est correctement réalisé sur un chemin de données commandé par une partie contrôle. Un grand pas doit être franchi entre la spécification (modèle du programmeur en assembleur) et l'implémentation (ensemble de modules électroniques). Le processeur est donc décrit à des niveaux d'abstraction successifs (langage assembleur, langage des micro-instructions, blocs électroniques), et la preuve montre la cohérence entre deux niveaux adjacents. Chaque étape de vérification doit établir qu'une opération d'un niveau abstrait est correctement réalisée par une séquence d'opérations plus simples à un niveau relativement plus concret.

1.2. *Objectifs et contributions*

Notre objectif est de fournir un système de vérification formelle des processeurs qui puisse être accepté par les concepteurs de circuits comme un complément à leurs outils de validation habituels. Pour cela le processus de spécification et celui de vérification ne doivent pas nécessiter de connaissance particulière en logique mathématique.

Nous avons d'une part conçu un environnement de spécification générique et convivial pour décrire les processeurs dans un style adapté à la preuve. D'autre part, la complexité des processeurs actuels et la différence d'abstraction entre la spécification et l'implémentation rendent la preuve des processeurs intraitable dans le cas général. Par conséquent, l'intervention de l'utilisateur est inévitable. Cependant, nous pensons qu'elle doit survenir le plus tard possible dans le processus de preuve et préconisons donc l'utilisation progressive de plusieurs méthodes :

— premièrement, des méthodes simples et efficaces comme la simplification (manipulation d'expressions symboliques en vue de les mettre sous une forme canonique).

Ces méthodes suffisent pour résoudre une grande partie des problèmes ;

— puis associer ces méthodes de simplification à une représentation compacte des données afin de retarder l'explosion combinatoire ;

— enfin, utiliser des méthodes complexes non automatiques.

Dans la suite de cet article, nous montrerons qu'une telle approche a permis de vérifier efficacement et avec un minimum d'intervention humaine, un ensemble d'exemples significatifs. En particulier nous avons mis en évidence des erreurs dans la spécification d'un processeur réel.

De plus, nous avons traité le cas des instructions implémentées par des boucles de micro-instructions, ce qui n'avait pas été fait auparavant dans le cadre d'un outil dédié à la preuve des processeurs.

1.3. *État de l'art*

La vérification formelle de circuits (voir [GUP 92] pour une présentation générale sur ce sujet) est maintenant utilisée en milieu industriel en ce qui concerne la preuve à bas niveau. Dans ce cadre, les outils utilisés sont basés sur des représentations efficaces des formules booléennes [BRY 86], et les preuves sont entièrement automatiques. Pour les niveaux plus abstraits, l'approche adoptée est d'utiliser des outils généraux de démonstration automatique tels que Nqthm [BOY 88], HOL [GOR 92] ou PVS [OWR 92].

En ce qui concerne le cas spécifique des processeurs, les premiers résultats ont été obtenus par Gordon [GOR 83], qui a vérifié un processeur simple avec le système LCF-LSM. Cette expérience a été suivie de deux cas d'études significatifs : le processeur VIPER a été en partie vérifié avec HOL [COH 87], et le FM8501 avec le démonstrateur Nqthm [Hun 89]. Dans les deux cas, les processeurs ont été décrits dans le langage formel associé au démonstrateur, et la spécification aussi bien que la preuve ont nécessité un grand savoir-faire.

L'étude de ces cas particuliers a mis en évidence la nécessité d'une méthodologie applicable à une classe générale de processeurs. Un modèle fonctionnel a d'abord été proposé dans [BOR 88] pour décrire les processeurs aux niveaux abstraits. Plus récemment, une méthodologie basée sur la notion d'interprète [ANC 86] générique a été proposée [JOY 90, WIN 90] pour la vérification des processeurs à tout niveau d'abstraction. Ce concept générique a été implémenté en HOL qui est, à notre avis, trop général pour être efficace et nécessite trop d'intervention humaine. Nous avons cependant adapté et étendu ce concept dans notre système de preuve (voir section 2).

Des résultats intéressants ont aussi été obtenus avec des méthodes plus spécialisées et plus automatiques [COU 89, BEA 94, BUR 94] ainsi qu'avec des approches hybrides [ZHU 93]. Toutefois, ces méthodes portaient sur des descriptions de bas niveaux et n'offraient pas d'interface de spécification.

La suite de cet article est organisée ainsi : nous détaillons l'environnement de description des processeurs dans la section 2, puis la méthodologie de preuve dans la

section 3. La section 4 présente les différents outils de preuve et la section 5 illustre leur utilisation coopérative sur un ensemble d'exemples significatifs.

2. Le processus de spécification

2.1. *Description des processeurs*

En enseignant l'architecture des ordinateurs, on introduit généralement un processeur comme un ensemble de composants (ces composants sont des registres, des bus, des multiplexeurs, ... Ils forment le chemin de données) qui interagissent en fonction des signaux émis par le séquenceur (la partie contrôle). Un processeur est ainsi défini par son état (ensemble des composants), ses transitions d'état (commandées par la partie contrôle à chaque top d'horloge) et son séquencement (la succession des transitions d'état).

Cela nous a amenés à décrire un processeur par un interprète ayant trois attributs :

— `state` : l'ensemble des composants visibles au niveau courant.

— `transitions` : les fonctions de transition d'état qui définissent le comportement du processeur,

— `select` : une fonction qui, selon l'état du processeur, renvoie la transition activée par la partie contrôle.

Un tel interprète est générique car il est réutilisable pour spécifier différents processeurs et s'applique à différents niveaux d'abstraction [WIN 90]. Ces niveaux sont, pour le plus abstrait, le processeur tel qu'il est vu par un programmeur en assembleur. Le niveau le plus concret est sa description structurelle sous forme d'un diagramme électronique.

2.2. *Description des preuves*

Pour assurer la généricité de la preuve, nous avons étendu le concept d'interprète générique. Nous décrivons les vérifications à effectuer entre deux niveaux de spécification par une entité désignant deux interprètes et définissant les fonctions d'abstraction de l'implémentation vers la spécification :

— `abstraction` : définit l'abstraction de l'état. Le plus souvent, elle est unique car il s'agit d'une projection : l'état de la spécification est un sous-ensemble de l'état de l'implémentation.

— `sync` définit l'abstraction temporelle. C'est un prédicat vrai seulement quand les deux niveaux sont synchronisés c'est à dire quand l'état de l'implémentation correspond au début d'une transition au niveau abstrait. Il est le plus souvent défini comme un test sur la valeur du compteur de programme de l'implémentation.

La preuve est effectuée en exécutant une transition au $niveau_i$ et une séquence de transitions au $niveau_{i+1}$. L'implémentation est correcte si l'état final au $niveau_i$ est

une abstraction de l'état final au $niveau_{i+1}$ (voir le diagramme de 3.1).

2.3. *Implémentation et interface utilisateur*

Nous avons choisi une approche objet pour implémenter l'environnement de spécification [ARD 95a]. Une telle approche est bien adaptée pour mettre en oeuvre les concepts de généricité et réutilisabilité [MEY 88, RUM 91].

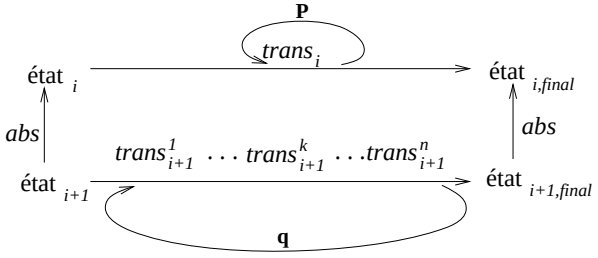
Chaque processeur est défini à chaque niveau comme un objet de la classe `Interpreter` qui a les trois attributs `state`, `transitions` et `select`. Chaque composant de l'attribut `state` est une instance d'une classe obtenue par héritage multiple. Elle hérite d'une classe structurelle (type de données) et d'une classe comportementale (comportement temporel) [ARD 95a]. De même une preuve est une instance de la classe `Proof` ayant les attributs `specification` (l'interprète représentant la spécification), `implementation` (l'interprète représentant l'implémentation), `abstraction` et `sync`. Cette hiérarchie de classes a été implémentée dans un langage fonctionnel orienté objet : STk [GAL 94] qui est un interprète Scheme associé à un système objet très proche de CLOS [Ste 90].

Pour aider les concepteurs qui ne sont pas familiers avec Scheme, nous avons conçu un langage simple pour décrire les processeurs dans un style facilitant la preuve. Sa syntaxe est proche du standard VHDL et nous avons montré dans [ARD 95b] la faisabilité de la traduction d'un sous-ensemble de VHDL vers ce langage. À partir de cette forme textuelle, un traducteur génère automatiquement la représentation interne Scheme.

3. Le processus de vérification

3.1. *Une approche coopérative*

Comme expliqué en 1.1, la vérification du jeu d'instruction d'un processeur par rapport à son architecture est réalisée par des équivalences successives entre des niveaux de spécification adjacents. Une étape de vérification consiste à montrer que le diagramme suivant commute :



où état_i est un niveau abstrait (par exemple, le langage assembleur), état_{i+1} est un niveau plus concret (par exemple, le langage des micro-instructions), abs est la fonction

d'abstraction, p et q représentent le nombre de pas dans les boucles (éventuellement nul).

Les principaux résultats déjà obtenus [Hun 89, JOY 90, WIN 90, STA 93] concernent le cas où $p = q = 0$. Nous proposons ici une approche coopérative entre plusieurs outils de preuve afin de résoudre également les autres cas.

Pour vérifier la commutation du diagramme ci-dessus, les mécanismes mis en jeu sont :

— l'abstraction : pour lier les deux visions du processeur,

— l'exécution symbolique : pour calculer les états symboliques définis par les transitions,

— la simplification et la composition : pour réduire une séquence de micro-opérations en une macro-opération et mettre les expressions sous forme canonique,

— l'induction : pour prouver des macro-opérations réalisées par des boucles de micro-opérations.

Cas où $p = q = 0$.

Ce cas décrit la plus grande partie du jeu d'instructions. Cela correspond à une instruction dont l'opération principale (par exemple une addition) est fournie par l'architecture. La différence entre les deux niveaux est simplement que le niveau concret est plus détaillé et nécessite plus de transferts intermédiaires. Dans ce cas, l'exécution symbolique et la simplification sont suffisants.

Par exemple, une instruction d'addition en mode d'adressage direct peut être spécifiée au niveau assembleur par :

$$\begin{array}{lllll} add: & REG[dest(MEM[PC])] & \leftarrow & REG[dest(MEM[PC])] + MEM[PC+1] \\ & PC & \leftarrow & PC + 2 \end{array}$$

PC est le compteur de programme, MEM la mémoire, $MEM[PC]$ désigne le code de l'instruction courante, $dest$ extrait le numéro du registre destination et REG est le banc des registres utilisateurs.

Au niveau des micro-instructions, cette instruction est implémentée par la séquence suivante :

$$\begin{array}{lllll} \mu_1: & IR & \leftarrow & MEM[PC] & \text{; lecture de l'instruction} \\ & PC & \leftarrow & PC + 1 & \\ \mu_2: & BUF & \leftarrow & MEM[PC] & \text{; utilisation d'un registre tampon} \\ & PC & \leftarrow & PC + 1 & \\ \mu_3: & REG[dest(IR)] & \leftarrow & REG[dest(IR)] + BUF & \end{array}$$

La preuve consiste à composer des transitions et à réaliser des simplifications symboliques.

Cas où $p \neq 0$ et $q \neq 0$.

Il s'agit d'instructions qui répètent p fois une opération, p étant la valeur d'un registre. Par exemple, l'instruction `rep stosb` du processeur Intel 8086, recopie p fois (p est la valeur du registre `cx`) le contenu du registre `al` en mémoire à partir de l'adresse contenue dans `di`. La spécification au niveau assembleur est :

$rep\ stosb$: tant que $cx \neq 0$ faire
$MEM[di] \leftarrow al$
$di \leftarrow di + 1$
$cx \leftarrow cx - 1$

Elle est implémentée par :

μ_1 : $ad \leftarrow di$; connecté au bus d'adresses
$da \leftarrow al$; connecté au bus de données
tant que $cx \neq 0$ faire
μ_2 : $MEM[ad] \leftarrow da$
$ad \leftarrow ad + 1$
$cx \leftarrow cx - 1$
fin
μ_3 : $di \leftarrow ad$

Une preuve inductive est indispensable puisque la condition de sortie de boucle teste la valeur d'un composant (cx) qui reste symbolique [1]. Dans un grand nombre de cas cependant, ce type de preuve ne demande pas d'intervention humaine, puisque chaque passage dans la boucle est du même ordre de complexité pour chacun des deux niveaux (l'opération élémentaire effectuée est fournie aux deux niveaux).

Cas où $p = 0$ et $q \neq 0$.

C'est le cas le plus critique. Il correspond en général à une instruction qui réalise une opération qui n'est pas fournie par le chemin de données et qui est donc réalisée par une boucle d'opérations élémentaires (opérations arithmétiques complexes, certains décalages et rotations).

Si la condition de sortie de boucle porte sur une valeur constante, l'exécution symbolique permet de calculer la valeur de l'état final. Toutefois, comme tous les passages dans la boucle sont tracés, on peut craindre une explosion combinatoire. On peut alors associer à l'exécution symbolique une représentation efficace des données. Ce cas est illustré par l'algorithme de multiplication par additions/décalages détaillé en 5.2. Pour un opérande de taille n, il faut effectuer n additions et décalages.

Par contre, si la condition de sortie de boucle porte sur une valeur symbolique, une preuve inductive est indispensable. C'est le cas par exemple d'une multiplication réalisée par un algorithme d'additions itérées où l'on additionne X fois la valeur de Y pour réaliser $X \times Y$. La preuve inductive peut nécessiter dans ce cas l'introduction de lemmes liant l'arithmétique du niveau abstrait et celle du niveau concret.

1. Sans preuve inductive, il faudrait tester 2^n cas si cx est un vecteur de n bits.

1. calculer un état initial S_{i+1} au niveau$_{i+1}$ tel que $sync(S_{i+1})$ est vrai :
    $S_{i+1} := init(state_{i+1})$
2. prendre une abstraction S_i de cet état :
    $S_i := abstraction(S_{i+1})$
s'il y a une boucle au niveau$_i$ (cas où $p \neq 0$) alors
    3. générer les fonctions récursives des niveau$_i$ et niveau$_{i+1}$:
        $Frec_i := generate\text{-}inductive(niveau_i)$
        $Frec_{i+1} := generate\text{-}inductive(niveau_{i+1})$
    4. montrer l'équivalence logique entre ces deux fonctions :
        $inductive\text{-}proof(Frec_i, abstraction(Frec_{i+1}))$

Figure 1. *L'algorithme de vérifi cation (début)*

3.2. *L'algorithme de preuve*

Supposons définis deux objets de la classe `Interpreter` qui spécifient un processeur aux niveaux $niveau_i$ (la spécification) et $niveau_{i+1}$ (l'implémentation), et un objet de la classe `Proof`, qui lie ces deux interprètes. La preuve entre les deux niveaux est réalisée par l'algorithme de les figures 1 et 2.

Les points 1 et 2 calculent des états initiaux à partir de valeurs symboliques.

Si l'instruction est une instruction de boucle, les fonctions récursives sont construites à partir de l'attribut `select` des interprètes (point 3). Le point 4 réalise une preuve inductive.

Les points 5 et 6 réalisent les transitions d'état ; « execute » calcule les transitions tout en simplifiant les expressions (sa sémantique est détaillée en 4.2.1). Le point 7 effectue une vérification syntaxique de l'égalité des deux expressions.

Si l'exécution symbolique échoue nous utilisons différentes techniques. Si l'échec est dû à une explosion combinatoire, nous essayons d'associer à l'exécution symbolique une représentation compacte sous forme de *BMDs (voir 4.3). Dans ce cas, le point 8 est équivalent aux points 1, 2, 5, 6 et 7 en utilisant les *BMDs comme représentation symbolique.

Si l'exécution symbolique échoue à cause d'une boucle dont la condition n'est pas évaluable ou si elle est dûe à une explosion combinatoire avec les *BMDs, le point 9 réalise une preuve inductive de l'équivalence des fonctions associées aux transitions. Celle du $niveau_i$ est primitive tandis que celle du $niveau_{i+1}$ est récursive. Cette preuve induira éventuellement une généralisation des expressions et l'introduction de lemmes ; elle ne sera donc pas automatique comme dans le point 4.

```
sinon
    5. exécuter la transition sélectionnée par l'interprète du niveau_i
    pour obtenir l'état final au niveau_i :
        trans := transitions(select(S_i))
        S_i := execute(trans, S_i)
    6. exécuter les transitions au niveau_{i+1}
    jusqu'à ce qu'une condition ne soit pas évaluable
    ou qu'il y ait une explosion combinatoire
    ou que les deux interprètes soient synchronisés :
        répéter
            trans := transitions(select(S_{i+1}))
            S_{i+1} := execute(trans, S_{i+1})
        jusqu'à non-évaluable(select(S_{i+1}))
    si sync(S_{i+1}) alors
        7. vérifier que  S_i ≡ abstraction(S_{i+1})
    fin si
    si explosion(S_{i+1}) alors
        8. utiliser une representation compacte (Diagrammes de Moments Binaires) :
            S_{i+1} := initBMD(state_{i+1})
            S_i := abstraction(S_{i+1})
            trans_i := transitions(select(S_i))
            S_i := execute(trans, S_i)
            répéter
                trans := transitions(select(S_{i+1}))
                S_{i+1} := execute(trans, S_{i+1})
            jusqu'à explosion(S_{i+1}) ou sync(S_{i+1})
            si sync(S_{i+1}) alors
                vérifier que  S_i ≡_BMD abstraction(S_{i+1})
            sinon aller en 9
            fin si
    fin si
    si non-évaluable(select(S_{i+1})) alors
        9. preuve inductive entre la spécification et l'implementation :
            Fprim_i := generate-primitive(niveau_i)
            Frec_{i+1} := generate-inductive(niveau_{i+1})
            inductive-preuve(Fprim_i, abstraction(Frec_{i+1}))
    fin si
fin si
```

Figure 2. *L'algorithme de vérifi cation (suite)*

usage	fonction
`concat(`$V_1, V_2, \ldots, V_n$`)`	concaténation
`field(`V, i, j`)` ou V`[`i`:`j`]`	sélection d'un champ de bits
`sum(`V_1, V_2`)`	somme d'un additionneur
`carry(`V_1, V_2`)`	retenue sortante d'un additionneur
`add(`V_1, V_2`)` équivalent à `concat(sum(`V_1, V_2`),carry(`V_1, V_2`))`	addition
`sub(`V_1, V_2`)`	soustraction
`incr(V)` équivalent à `sum(V,"1")`	incrémentation
`and(`$V_1, V_2, \ldots, V_n$`)`	opérations logiques bit-à-bit
`not(`V`)`	négation logique
`mux(`C, V_1, V_2`)` équivalent à `if` C `then` V_1 `else` V_2	sortie d'un multiplexeur
`BV_nat(`V`)` équivalent à $\$V$	conversion vers les entiers naturels
`exts(`V, n`)`	extension à n bits
`length(`V`)`	longueur d'un vecteur de bits
`lsb(`V`)` équivalent à V`[0:0]`	bit de poids le plus faible
`msbs(`V`)` équivalent à V`[1:length(`V`)-1]`	bits de poids forts
`BV_null(`n`)`	vecteur formé de n bits nuls
`eq(`V_1, V_2`)` ou $V_1 = V_2$	égalité

Tableau 1. *L'algèbre des vecteurs de bits*

4. Les techniques de preuve

Avant de présenter les différentes techniques de preuve, nous introduisons l'algèbre de vecteurs de bits à la base de toutes ces techniques.

4.1. *Algèbre de vecteurs de bits*

La vision abstraite d'un processeur opère sur des valeurs entières qui sont codées au niveau concret par des vecteurs de bits. Nous devons donc définir une algèbre sur les vecteurs de bits qui inclut les opérations logiques usuelles : concaténation, décalage, rotation, et logique, . . .

Puisque notre but est de prouver l'équivalence entre les deux vues du processeur, nous devons aussi fournir des fonctions d'abstraction des données entre les entiers naturels et les vecteurs de bits. Le tableau 1 décrit les principaux opérateurs de cette algèbre.

4.2. *Exécution symbolique et simplification*

La technique la plus simple que nous utilisons est l'exécution symbolique associée à un système de simplification que nous détaillons ici.

4.2.1. *Un évaluateur symbolique de transitions*

Le comportement des processeurs est défini par des fonctions de transition d'état. Soit une transition $dest \leftarrow source$; son exécution se déroule comme suit :

— un appel à la méthode `read` rend la valeur de $source$ si elle désigne ou inclut des composants du processeur,

— le système de simplification est appelé pour réduire la valeur rendue sous une forme canonique,

— la méthode `write` est appelée pour modifier la valeur de $dest$.

L'évaluateur symbolique tire parti de l'approche orientée-objet : le processus d'évaluation est générique du fait que les méthodes `read` et `write` sont associées à chaque classe de composants. C'est par discrimination que la méthode correspondant au type du composant auquel on accède est sélectionnée.

4.2.2. *Le système de simplifi cation*

Tout système de vérification formelle basé sur les techniques de preuve utilise comme outil de base un moteur de simplification symbolique. En effet, les preuves doivent manipuler les valeurs symboliques en entrée des spécifications. Comme deux niveaux d'un même système sont décrits par des opérations différentes, les valeurs finales symboliques diffèrent par leur syntaxe. Il faut alors vérifier qu'elles sont sémantiquement équivalentes. Or cette vérification sémantique peut être réalisée par une mise sous forme canonique, puis par une vérification syntaxique. Pour ce faire, nous avons développé un système de calcul formel dédié à la manipulation des entiers et des vecteurs de bits. Ce système est moins puissant que des démonstrateurs de théorèmes généraux comme HOL, Nqthm, PVS, Larch Prover, OBJ3 . . . mais il est le plus souvent suffisant pour la vérification des processeurs. De plus il est bien plus efficace et facile d'utilisation : il est entièrement automatique et plus rapide qu'un système plus général. Les résultats que nous avons obtenus et qui sont donnés en 5.1 confirment cette affirmation. Il est particulièrement bien adapté pour ce problème spécifique où les expressions sont simples mais très nombreuses, et pour lequel l'efficacité et la simplicité doivent supplanter la puissance et la généralité.

D'autres systèmes de preuve utilisent des techniques de réécriture pour simplifier des expressions symboliques (voir par exemple [SEK 89, STA 93]). Nous avons choisi une approche basée sur le calcul formel par souci d'efficacité et pour traiter facilement les opérateurs associatifs et commutatifs.

4.3. **BMD : une représentation symbolique compacte des vecteurs de bits*

La seconde technique de preuve que notre système utilise est aussi basée sur l'exécution symbolique mais s'appuie sur une représentation plus efficace.

4.3.1. *Pourquoi les *BMDs ?*

Certaines instructions complexes réalisent des opérations qui ne sont pas implémentées par un module du chemin de données et sont donc réalisées par des algorithmes qui contiennent des boucles. Si la condition de sortie de boucle porte sur une valeur non symbolique (i.e., dépend d'un composant qui a été affecté avec une constante), l'exécution symbolique peut être utilisée pour tracer toutes les étapes. Cependant une représentation compacte des vecteurs de bits est essentielle pour éviter, ou au moins retarder l'explosion combinatoire provoquée par ces boucles. De plus, cette technique de preuve reste entièrement automatique.

Parmi les nombreuses représentations de formules booléennes, nous avons choisi les «Binary Moment Diagrams» (*BMDs) pour deux raisons majeures : ils fournissent une représentation canonique des fonctions de $\{0,1\}^n$ dans $\mathbb{Z}$ [BRY 95]. De plus, les représentations des vecteurs de bits et de l'addition ont une taille qui croit linéairement avec le nombre de bits. C'est exactement ce que nous demandons pour traiter des instructions complexes qui implémentent des opérations arithmétiques sur les vecteurs de bits.

4.3.2. *Définition des *BMDs*

Les *BMDs héritent des concepts élaborés pour les «Binary Decision Diagrams» (BDDs) : ils représentent des fonctions sur des variables booléennes par des graphes acycliques. Soit f une fonction de $\{0,1\}^n$ dans $\mathbb{Z}$. Les BMDs sont basés sur l'expansion de Reed-Muller de f par rapport à une variable booléenne x. Nous notons $f_{\overline{x}}$ la restriction de f quand $x = 0$ et f_x sa restriction quand $x = 1$. L'expansion de Shannon de f par rapport à x donne tout d'abord [2]

$$\begin{aligned} f &= (1 - x) \cdot f_{\overline{x}} + x \cdot f_x \\ &= f_{\overline{x}} + x \cdot (f_x - f_{\overline{x}}) \end{aligned}$$

f est ainsi décomposée en deux moments :

$$f = f_{\overline{x}} + x \cdot f_{\dot{x}} \qquad \text{où} \qquad f_{\dot{x}} = f_x - f_{\overline{x}}$$

$f_{\overline{x}}$ est le moment constant. Il représente la valeur de la fonction quand $x = 0$. L'autre moment, $f_{\dot{x}}$, est le moment linéaire. Il représente la variation de f quand x passe de 0 en 1. Un BMD est donc un arbre avec un identificateur de variable comme racine et deux BMDs comme fils qui sont les moments constant et linéaire.

Les BMDs ont des valeurs numériques associées aux feuilles mais aussi aux branches. Ces valeurs sont multipliées au cours du parcours d'un arbre ; cette structure est donc appelée Multiplicative Binary Moment Diagram (*BMD). Comme pour les autres représentations symboliques équivalentes, les tailles des *BMDs sont réduites en partageant des sous-arbres communs.

Bryant et Chen ont proposé des fonctions de synthèse en appliquant l'addition, la multiplication ou l'exponentiation à des *BMDs tout en conservant une forme cano-

---

2. $+, -, \cdot$ représentent l'addition, la soustraction et la multiplication pour entiers.

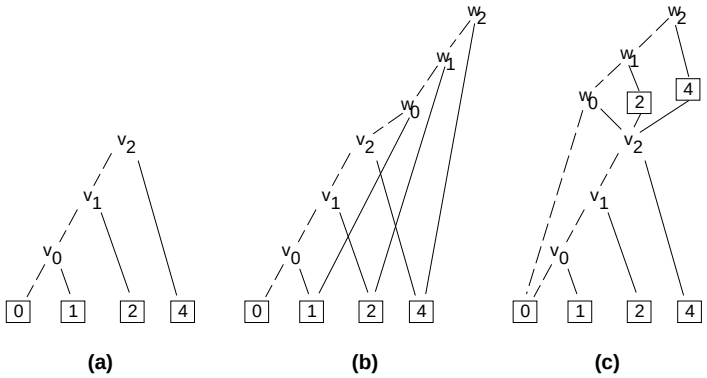


Figure 3. *Les représentations *BMD d'un mot non signé* **(a)**, *de l'addition* **(b)** *et de la multiplication* **(c)**. *Les valeurs des branches égales à* 1 *ne sont pas indiquées, les lignes pointillées sont les moments constants, les lignes pleines les moments linéaires.*

nique [BRY 94]. La figure 3 montre les représentations de quelques opérations avec des *BMDs. Par exemple, on peut voir en **(a)** que la valeur non signée du vecteur composé des bits v_0 (poids faible), v_1 et v_2 est $4 * v_2 + (2 * v_1 + (1 * v_0 + 0)) = 4 * v_2 + 2 * v_1 + v_0$. La figure 3 **(c)** illustre le produit des deux vecteurs W composé de w_0, w_1, w_2 et V composé de v_0, v_1, v_2. La fonction ainsi représentée vaut $4*w_2*(4*v_2+2*v_1+v_0)+2*w_1*(4*v_2+2*v_1+v_0)+w_0*(4*v_2+2*v_1+v_0) = (4 * w_2 + 2 * w_1 + w_0) * (4 * v_2 + 2 * v_1 + v_0)$ ce qui est bien la valeur de $W * V$.

Soit par exemple la fonction $f(v_2, v_1, v_0) = 4v_2 + 2v_1 + v_0$. La décomposition de f par rapport à v_2 donne les deux moments :

$$f_{\overline{v_2}} = 2v_1 + v_0 \quad \text{et} \quad f_{v_2} = 4$$

L'évaluation de f se déroule comme suit : si $v_2 = 0$, alors on rend le résultat de l'évaluation de $2v_1 + v_0$, sinon on rend la somme de l'évaluation de $2v_1 + v_0$ et de 4.

4.3.3. *Une représentation des vecteurs de bits par les *BMDs*

Nous avons étendu les *BMDs dans le but de fournir tous les opérateurs de l'algèbre décrite en 4.1 [ARD 96]. Nous utilisons les *BMDs exactement comme s'ils représentaient des vecteurs de bits et non pas des entiers. Pour cela nous devons tenir compte de la taille des vecteurs ; nous complétons donc le concept des *BMDs comme suit. Un vecteur de s bits est représenté par un *BMD$_{(s)}$ qui est un couple formé par un *BMD et l'entier s. Nous notons $Z_{(s)}$ pour signifier que Z est la représentation en *BMD de la valeur non signée d'un vecteur de s bits [3]. Alors que les *BMDs représentent des fonctions de $\{0,1\}^n$ vers $\mathbb{Z}$, les *BMD$_{(s)}$ représentent les fonctions de $\{0,1\}^n$ vers les entiers de $[0; 2^s[$. Nous notons *BMD$_{()}$ l'union de tous les *BMD$_{(s)}$ pour $s > 0$.

3. s n'est pas le nombre de variables de Z mais il est le nombre minimum de bits requis pour représenter toutes les valeurs possibles de Z.

L'opérateur le plus simple que nous avons défini est la concaténation.

L'opérateur $concat$ rend la concaténation de deux *BMD$_{()}$:

$$concat(Z_{(s)}, Z'_{(s')}) \stackrel{def}{=} (Z + 2^s * Z')_{(s+s')}$$

Les autres opérateurs sur les vecteurs de bits sont définis en utilisant la structure inductive des vecteurs (assimilables à des listes) : un vecteur est composé d'un bit de tête et d'un vecteur de bits formant sa queue. Ces opérateurs ne peuvent pas être directement définis par une induction sur la structure en graphe des *BMDs.

Ainsi, nous avons tout d'abord défini trois primitives pour reproduire la structure des vecteurs avec les *BMD$_{()}$.

La primitive $cons$ construit un vecteur à partir d'un bit et d'un vecteur. Elle prend en argument un *BMD$_{(1)}$ représentant un bit et un *BMD$_{(s)}$ représentant un vecteur de bits. Le résultat est un *BMD$_{(s+1)}$ tel que :

$$cons(B_{(1)}, V_{(s)}) \stackrel{def}{=} (B + 2 * V)_{(s+1)}$$

La primitive lsb rend le bit de poids faible d'un vecteur. Nous ne traitons que les mots positifs pour lesquels le bit le moins significatif est à 1 quand la valeur du mot est impaire ; ce bit est à 0 quand la valeur est paire. Le résultat de la fonction lsb dépend donc de la parité du *BMD$_{(s)}$, avec $s > 0$:

$$s > 0 \ \rightarrow \ lsb(V_{(s)}) \stackrel{def}{=} (odd(V))_{(1)}$$

La fonction odd sur les *BMDs découle directement des propriétés de la parité vis-à-vis de la multiplication et de l'addition. Voici sa définition :

$odd($*BMD terminal de valeur $w) \stackrel{def}{=} w$ modulo 2

$$odd(V \text{ de valeur } w) \stackrel{def}{=} \begin{cases} 0 \text{ si } (w \text{ modulo } 2) = 0 \\ odd(V_{\overline{x}}) + x \cdot (odd(V_{\dot{x}}) - 2 * odd(V_{\overline{x}}) * odd(V_{\dot{x}})) \text{ sinon} \end{cases}$$

La première équation rend un *BMD terminal dont la valeur représente la parité du *BMD argument quand il est terminal. La seconde équation indique qu'un *BMD dont la valeur de la racine est paire est toujours pair car le produit d'un nombre pair avec une valeur quelconque est toujours pair. Si la valeur de la racine est impaire, alors le résultat est un *BMD non terminal calculé par des appels récursifs à odd. Ce résultat est obtenu en constatant que si $x = 0$, alors la parité est celle de $V_{\overline{x}}$. Si $x = 1$, la parité est celle de la somme $V_{\overline{x}} + V_{\dot{x}}$. Cette somme est impaire quand $V_{\overline{x}}$ et $V_{\dot{x}}$ sont tous les deux impairs. On a ainsi $odd(V_{\overline{x}} + V_{\dot{x}}) = V_{\overline{x}} + V_{\dot{x}} - 2 * V_{\overline{x}} * V_{\dot{x}}$.

La primitive $msbs$ rend les bits de poids fort d'un vecteur. Elle est construite en utilisant le même schéma récursif que pour lsb.

Grâce à ces trois primitives, on peut définir tous les opérateurs de l'algèbre donnée en 4.1. Ils sont utilisés au point 8 de l'algorithme de preuve (voir 3.2) pour calculer

les *BMD qui représentent les états du processeur aux deux niveaux. Nous donnons un exemple ci-dessous.

La sélection d'un champ de bits est récursivement définie. $field(Z_{(s)}, i, j)$ extrait les bits de $Z_{(s)}$ dont les positions sont entre i et j :

$$s > 0 \;\rightarrow\; field(Z_{(s)}, 0, 0) \stackrel{def}{=} lsb(Z_{(s)})$$

$$s > j > 0 \;\rightarrow\; field(Z_{(s)}, 0, j) \stackrel{def}{=} cons(lsb(Z_{(s)}), field(msbs(Z_{(s)}), 0, j-1))$$

$$s > j \geq i > 0 \;\rightarrow\; field(Z_{(s)}, i, j) \stackrel{def}{=} field(msbs(Z_{(s)}), i-1, j-1)$$

4.4. *Preuves inductives avec l'assistant de preuve Coq*

Nous présentons dans cette section la dernière technique utilisée. Elle réalise des preuves inductives et demande généralement une intervention humaine. C'est pourquoi cette technique est utilisée en dernier choix, quand l'exécution symbolique, avec ou sans les *BMDs, échoue.

4.4.1. *Pourquoi Coq ?*

Coq est un assistant de preuve [COR 95] pour le λ-calcul d'ordre supérieur. Ses bases théoriques sont le Calcul des Constructions [COQ 88] étendu avec des définitions inductives [COQ 90]. Le développement d'une preuve correspond à la construction d'un λ-terme ; cette caractéristique est exploitée par la possibilité de synthétiser un programme à partir de la preuve de sa spécification. Le lecteur est invité à se référer à [COR 95] pour une description détaillée de ce système.

Nous avons déjà justifié le besoin de preuves inductives pour vérifier les instructions de boucles dépendantes des données (voir le diagramme de 3.1). Nous donnons ici les raisons qui nous ont fait choisir Coq.

La première raison est que, de notre point de vue, une preuve inductive est la dernière méthode à utiliser car elle demande une intervention humaine. Nous avons donc choisi un assistant de preuve et pas un démonstrateur automatique. L'avantage est que la preuve est réalisée pas à pas par l'utilisateur qui a un contrôle total. À l'opposé, avec un démonstrateur automatique comme Nqthm, il est très difficile de comprendre pourquoi une preuve échoue et de la mener à bien dans ces cas. La seconde raison est que nous avons besoin d'un système basé sur une logique d'ordre supérieur. En effet, dans la modélisation des circuits séquentiels, les composants sont représentés par des fonctions sur le temps. Lorsque l'on décrit un système complet par les connections de ces composants, ces fonctions sont des arguments d'autres fonctions.

Coq répond à ces exigences et des travaux précédents ont montré qu'il peut être utilisé en milieu industriel pour des vérifications à des niveaux abstraits ([BOL 95]).

4.4.2. *Des théories pour la vérifi cation des circuits avec Coq*

La vérification des circuits semble être une des applications majeures des démonstrateurs de théorèmes et des assistants de preuve. Mais ce n'est pas le cas de Coq puisque, à notre connaissance, les uniques travaux réalisés dans ce domaine avec Coq sont dûs à Coupet-Grimal et Jakubiec [COU 96] et Paulin-Mohring [PAU 95] ; ils concernent uniquement des circuits combinatoires ou séquentiels simples décrits au niveau des entiers. Comme nous utilisons Coq dans un environnement plus général, nous avons tout d'abord dû implémenter tous les types de données utilisés dans notre algèbre : les vecteurs de bits et les mémoires. Les booléens (`bool`, `false`, `true`) et les entiers naturels (`nat`, `0`, `S`, `plus`, `minus`, `mult`) sont modélisés en utilisant les théories standards fournies dans la distribution de Coq.

Nous utilisons la théorie de Coq qui définit les listes polymorphes. Ces listes sont définies par le type `list` qui a comme constructeurs `nil` et `cons` pour lesquels le premier argument est le type des éléments.

Les principales fonctions prédéfinies sur les listes sont la concaténation (`app`, `concat`) et la longueur (`length`). Nous avons défini un certain nombre de fonctions supplémentaires (renversement, sélection d'une sous-liste, . . .). De nombreuses propriétés concernant ces fonctions ont été prouvées. La grande expressivité de Coq nous a laissé une grande liberté dans la formulation de ces fonctions mais son manque d'automatisme s'est ressenti lors des preuves des propriétés. Cependant, cet inconvénient est relatif puisque ce travail n'a pas à être répété : la théorie complète définissant les listes polymorphes, les fonctions associées et leurs propriétés est « compilée » et peut être réutilisée.

Ainsi, pour définir le type des vecteurs de bits, BV, il suffit d'instancier le type `list`. En effet, les vecteurs de bits sont des listes dont les éléments sont des bits, un bit étant représenté par un booléen. De même, une mémoire est une liste, de longueur fixe, de vecteurs de bits ayant tous la même taille. On réutilise donc le type des listes polymorphes en instanciant le type des éléments par BV. Les opérateurs de l'algèbre présentée en 4.1 sont des instanciations des fonctions définies pour les listes polymorphes.

L'abstraction de données, réalisée par une conversion des vecteurs de bits vers les entiers naturels, [4] est définie par la fonction `BV_nat`.

D'autre part, nous avons axiomatisé en Coq certains opérateurs souvent utilisés comme modules primitifs. Par exemple, l'addition est définie par une induction structurelle sur les vecteurs arguments et la correction des opérateurs `sum` et `carry` s'énonce par exemple comme suit :

```
∀ V,W:BV  BV_nat(concat(sum(V,W),carry(V,W)))
          = BV_nat(V)+BV_nat(W)
```

Les définitions du type BV, de ces fonctions et théorèmes relatifs sont encapsulées dans un module « compilé ». Ce module est en fait très proche de la bibliothèque

4. La fonction inverse – la conversion des naturels vers les vecteurs de bits – n'est pas utile car, dans notre méthodologie de vérifi cation, nous réalisons uniquement une abstraction d'un niveau concret (décrit par des vecteurs de bits) vers un niveau abstrait (décrit par des vecteurs de bits ou des naturels).

`word` écrite pour HOL [WON 93]. Il est important de noter que les théorèmes que nous avons prouvés sur les vecteurs de bits correspondent aux règles de simplification de notre système de calcul formel. Le développement de la théorie `BV` est donc une validation partielle du noyau de simplification de notre système.

4.4.3. *Modélisation des microprocesseurs*

Il a été montré que la logique d'ordre supérieur est un bon formalisme pour spécifier et vérifier les circuits digitaux [GOR 85]. La vérification formelle avec Coq peut donc bénéficier de ces idées.

Nous modélisons l'état d'un processeur par un ensemble de fonctions donnant la valeur des composants. Les variables modifiées à l'intérieur de boucles sont représentées par des fonctions récursives sur le « nombre de passages dans la boucle ». Cette valeur est nulle à l'entrée de la boucle ; elle est incrémentée à chaque passage. Cette approche est assez voisine de celle adoptée avec Nqthm dans [Hun 89]. Coq permet les définitions de fonctions mutuellement récursives, contrairement aux autres systèmes. On peut ainsi définir une fonction par composant alors que les autres travaux définissent une seule fonction qui manipule l'état global sous forme d'un n-uplet. Notre approche est plus facile à spécifier, plus lisible et les preuves sont souvent plus aisées.

Soient $R_1, \ldots, R_n$ les registres et mémoires d'un processeur. Chaque R_i est une fonction définie comme suit :

$$R_i(0) = init_i$$
$$R_i(t+1) = update_i(R_1(t), \ldots, R_n(t))$$

où $init_i$ est la valeur initiale du composant, généralement la valeur d'une entrée ou une valeur «don't care» (une valeur symbolique). À l'instant $t+1$, la valeur du composant est calculée par $update_i$ en utilisant les valeurs de tous les composants à l'instant précédent. Nous voyons ici l'importance des fonctions mutuellement récursives.

Comme nous voulons limiter l'intervention humaine, nous avons défini des algorithmes pour générer automatiquement les formes récursives présentées ci-dessus. Cela revient à construire le graphe de contrôle de l'implémentation et à réaliser deux parcours de ce graphe pour détecter les cycles et construire les fonctions.

Une preuve d'une instruction de boucle consiste à montrer qu'un prédicat, $correc$, sur les valeurs des composants et/ou des sorties, est vrai quand une condition, $cond$, est vérifiée :

$$\forall t \geq 0, \quad cond(R_1(t), \ldots, R_n(t)) \rightarrow correc(R_1(t), \ldots, R_n(t))$$

Intuitivement, ce théorème indique qu'à l'instant où la condition de terminaison du programme est vraie, le résultat est correct.

La preuve demande souvent une généralisation, particulièrement quand $correc$ est une fonction primitive comme la multiplication. L'utilisateur doit trouver un invariant de boucle et le prouver. Cet invariant est un prédicat, inv, sur les valeurs des registres

Processeur	instr.	μinstr.	registres	taille des mots	modes d'adressage
Tamarack-3	8	16	3	16	1
Proc. Anceau	8	14	4	16	4
DP-32	20	10	258	32	2
AVM-1	30	51	35	32	1
MTI	22	149	38	16	5

Tableau 2. *Tailles des processeurs que nous avons vérifi és*

qui est inconditionnellement vérifié :

$$\forall t \geq 0, \quad inv(R_1(t), \ldots, R_n(t))$$

Ce théorème indique qu'à tout instant un résultat partiel est correct. Le théorème final n'est alors qu'une spécialisation de cet invariant.

5. Résultats expérimentaux

Dans cette section, nous donnons plusieurs exemples de vérifications et résultats. Les premiers concernent les vérifications de processeurs complets en utilisant l'exécution symbolique et la simplification. Ensuite nous détaillons la vérification d'une instruction de multiplication en utilisant les *BMDs. Pour illustrer les preuves inductives avec Coq, nous revérifions la même instruction ainsi qu'une instruction de manipulation de chaînes.

5.1. *Vérification de microprocesseurs avec exécution symbolique et simplification*

Les processeurs considérés ici ont été spécifiés avec notre environnement orienté objet et vérifiés de façon automatique par exécution symbolique et simplification (points 5, 6 et 7 de l'algorithme de preuve). L'utilisateur est donc intervenu uniquement lors de la spécification. La complexité de ces preuves est illustrée dans le tableau 2. Le tableau comparatif 3 montre les tailles des spécifications et les temps de preuve pour un certain nombre de processeurs déjà vérifiés avec d'autres méthodes.

Le processeur Tamarack-3 et celui donné en exemple dans l'ouvrage de Anceau [ANC 86] (pages 181–212) sont des exemples d'école. Leurs spécifications et preuves ne posent aucun problème. Il est intéressant de comparer les résultats pour Tamarack-3. Stavridou [STA 93] a utilisé le démonstrateur OBJ3 qui est basé sur la réécriture. On peut voir que le temps de preuve n'est pas acceptable. Cyrluk et al. [CYR 94] ont utilisé PVS avec l'aide d'un simplifieur d'expressions booléennes qui utilise des BDDs. Leur temps de preuve est du même ordre que le notre, mais ils ne proposent pas un environnement générique. Le temps de preuve de [ZHU 93] est aussi satisfaisant mais, à notre connaissance, les spécifications sont données en HOL, un formalisme peu approprié pour la conception de circuits.

Processeur	#SL	SS (p.)	PT (sec.)	ref SS	ref PT	ref
Tamarack-3	4	3	197	17	10 jours	[STA 93]
				?	545 sec.	[CYR 94]
				?	360 sec.	[ZHU 93]
Proc. Anceau	2	4	183			
DP-32	3	12	435			
AVM-1	4	22	1775	110	58 h	[WIN 90]
MTI	2	17	1973			

Tableau 3. *Processeurs spécifiés et prouvés. #SL est le nombre des différents niveaux d'abstraction, SS est une taille approximative des spécifications en nombre de pages, PT est le temps de preuve en secondes sur un SUN IPC.*

Les autres processeurs sont plus complexes et leurs preuves montrent l'utilité de notre méthodologie de spécification et de vérification. DP-32 est un processeur décrit en VHDL dans [ASH 90]. Sa vérification a détecté une erreur dans son implémentation. Ce travail nous a permis de dégager un sous-ensemble de VHDL et un style de description dans ce langage qui est facilement traduisible dans le langage de spécification de notre système [ARD 95b]. AVM-1 a déjà été vérifié par Windley avec HOL [WIN 90]. Notre spécification est plus concise et lisible ; notre temps de preuve est aussi plus satisfaisant [5]. MTI est un processeur réel conçu par le CNET [PUL 87]. Il a déjà été étudié [BOR 88] mais jamais complètement vérifié. Notre système a découvert plusieurs erreurs dans son implémentation.

5.2. *Vérification d'une instruction de multiplication avec les *BMDs*

Pour illustrer les principes de la vérification avec des *BMDs, nous détaillons ici la vérification d'une instruction de multiplication.

Spécification et vérification

Nous détaillons seulement une instruction de multiplication et laissons ici de côté les autres aspects de la preuve. Nous considérons quatre registres de n bits (n étant une constante) : X, Y qui conservent les deux entrées, et R_l, R_h dans lesquels le résultat est construit (leur concaténation forme le résultat final). La sémantique de l'instruction est :

$$\begin{array}{lccl} multi: & R_l & \leftarrow & \texttt{field}(X \times Y,\ 0,\ n-1) \\ & R_h & \leftarrow & \texttt{field}(X \times Y,\ n,\ 2n-1) \end{array}$$

Le micro-programme de cette instruction implémente l'algorithme classique d'ad-

5. Notons toutefois que HOL a évolué depuis 1990 et que le temps de preuve avec ce système est certainement maintenant inférieur à celui donné ici.

n	taille finale	taille max.	temps (s)	temps GC (s)
4	9	15	0.266	0.000
8	17	45	1.150	0.000
16	33	153	4.433	0.000
32	65	561	38.333	6.433
64	129	2145	359.900	73.133
128	257	8385	3203.166	746.500

Tableau 4. *Résultats de la vérifi cation du micro-programme de multiplication*

ditions/décalages :

μ_1 : $R_l \leftarrow X$
$R_h \leftarrow$ `BV_null(`n`)`
$C \leftarrow n$
while $C > 0$ **do**
μ_2 : $R_l \leftarrow$ `concat(msbs(`R_l`),`
`mux(lsb(`R_l`),lsb(sum(`R_h, Y`)),lsb(`R_h`)))`
$R_h \leftarrow$ `concat(msbs(mux(lsb(`R_l`),sum(`R_h, Y`),`R_h`)),`
`mux(lsb(`R_l`),carry(`R_h, Y`),BV_null(1)))`
$C \leftarrow C - 1$

Cette description n'utilise que les opérateurs fournis dans notre algèbre de vecteurs de bits. Elle est donc totalement indépendante de la technique de vérification utilisée (simplification, *BMDs, ou preuve inductive avec Coq).

Comme le test de sortie de boucle porte sur C qui a une valeur constante, l'exécution symbolique est utilisée. Toutefois, avec le système de simplification, une explosion combinatoire se produit dès que $n > 3$. Nous avons donc recours au point 8 de l'algorithme de preuve (voir section 3.2). Ici encore, l'utilisateur intervient uniquement lors de la spécification.

Résultats et discussion

La vérification de cette instruction de multiplication est rapide. Le tableau 4 montre la taille finale du *BMD (en nombre de noeuds) pour différentes tailles de mots (n). La troisième colonne montre la taille du plus grand *BMD qui est généré durant la vérification. Cela donne une idée de la place mémoire nécessaire. Dans la quatrième colonne, nous indiquons le temps d'exécution nécessaire à la vérification complète. Cela inclut l'exécution symbolique, la synthèse des *BMDs et les « glanages de cellules » (récupération de mémoire). La dernière colonne donne les temps de garbage collection. Les expériences ont été menées sur un Sun SparcStation 10 avec un interprète de Scheme.

On peut voir sur ce tableau que la taille finale croît en $O(n)$, que la taille maximale est en $O(n^2)$ et le temps de preuve en $O(n^3)$.

La comparaison avec des travaux similaires n'est pas facile puisque, à notre connaissance, un tel microprogramme n'a été vérifié qu'en utilisant un démonstrateur de théo-

rèmes dans [SCH 94]. Leur spécification et implémentation ressemblent aux notres mais leur vérification demande beaucoup d'intervention humaine, surtout pour trouver un invariant correct.

De nombreux multiplieurs ont été vérifiés en utilisant des BDDs ou des structures dérivées des BDDs. Les meilleurs résultats ont été obtenus grâce à des optimisations particulières [BUR 91, BER 95]. Nos résultats sont meilleurs car les BDDs ne sont pas une bonne représentation pour la multiplication [BRY 86] : ils ont une taille exponentielle alors que les *BMDs ont une taille linéaire.

Bryant et Chen ont vérifié des multiplieurs combinatoires avec des *BMDs en utilisant une méthodologie de vérification hierarchique proposée dans [LAI 92]. Ils prouvent d'abord que chaque composant est correct vis-à-vis de sa spécification. Ceci est réalisé en utilisant des BDDs et une conversion des BDDs vers les *BMDs. Ensuite ils prouvent que la composition des composants est correcte par rapport à la spécification du multiplieur donnée au niveau des entiers. Leurs temps de preuve sont légèrement meilleurs que les notres parce qu'ils réalisent une vérification hiérarchique qui n'est pas totalement automatique.

Autres résultats

Nous avons appliqué notre méthodologie de vérification avec les *BMDs à des microprogrammes qui implémentent d'autres instructions arithmétiques : l'élévation au carré, l'exponentiation et la division. Ces expériences ont révélé la spécificité des *BMDs : ils ont une taille raisonnables pour la multiplication et l'élévation au carré. Mais leur utilisation pour vérifier l'exponentiation et surtout la division mène rapidement à une explosion combinatoire (il est difficile de vérifier une division pour des mots plus grands que 6 bits). Cette explosion semble être dûe au fait que le microprogramme de division contient un test comparatif entre deux *BMDs : $f < g$ qui est transformé en $f - g < 0$. Mais trouver les solutions d'un tel test est difficile car les *BMDs ne sont pas une représentation « facilement inversible » [BRY 94].

5.3. *Vérification d'une instruction de multiplication avec Coq*

Nous détaillons ici la vérification de la même instruction de multiplication mais en utilisant l'assistant de preuve Coq. Ceci met en évidence la difficulté à mener une preuve avec un démonstrateur général. En effet, l'utilisateur doit intervenir pour introduire les lemmes nécessaires à la preuve et pour trouver la forme sous laquelle le théorème peurra être prouvé.

Formalisation en Coq

Nous commençons par déclarer quelques variables et axiomes. Comme la taille des mots n'est pas connue, nous la définissons comme une variable globale `size` de type `nat` mais sans valeur particulière. Nous posons juste l'axiome indiquant que cette taille n'est pas nulle. Les deux registres `X` et `Y` ont des valeurs constantes durant

tout le calcul, nous les déclarons donc comme des vecteurs de bits dont la taille est `size` [6].

`size : nat` avec `size≠0`

`X, Y : nat` avec `length(X)=size` et `length(Y)=size`

Comme indiqué dans la section 4.4.3, les valeurs de `Rl` et `Rh` sont modélisées par des fonctions mutuellement récursives sur le temps :

`Rl(0)` $\stackrel{def}{=}$ `initl` et `Rl(t+1)` $\stackrel{def}{=}$ `updatel(Rl(t),Rh(t))`

`Rh(0)` $\stackrel{def}{=}$ `inith` et `Rh(t+1)` $\stackrel{def}{=}$ `updateh(Rl(t),Rh(t))`

A l'instant `0` le registre `Rl` a la valeur `initl` et à l'instant `t+1` la valeur `updatel(Rl(t),Rh(t))`. `Rh` est défini de la même manière. Nous devons maintenant spécifier les valeurs initiales et les fonctions de modifications :

```
initl       =def X
updatel(A,B) =def concat(msbs(A),mux(lsb(A),lsb(sum(B,Y)),
                         lsb(B)))
inith       =def BV_null(size)
updateh(A,B) =def concat(msbs(mux(lsb(A),sum(B,Y),B)),
                         mux(lsb(A),carry(B,Y),BV_null(1)))
```

Le compteur de boucle dépend uniquement du temps :

`C(0)` $\stackrel{def}{=}$ `size` et `C(t+1)` $\stackrel{def}{=}$ `C(t)-1`

Vérifi cation

Avant de prouver la correction fonctionnelle du micro-programme, nous devons vérifier quelques propriétés essentielles :

— La valeur du compteur est toujours égale à `size-t`.

— Le compteur atteint la valeur `0` quand celle du temps est `size`. Ceci montre que le micro-programme termine.

— Les tailles des valeurs des registres sont toujours égales à `size`.

— Il est aussi nécessaire de montrer que $\forall t <$ `size`, le bit le moins significatif de `Rl` est le $t^{ème}$ bit de `X`.

Enfin, le théorème final montre que quand le compteur de boucle atteint `0`, la concaténation de `Rl` et `Rh` est égale au produit de `X` et `Y` (par des abstractions) :

```
∀ t:nat  C(t)=0 → t≤size → BV_nat(concat(Rl(t),Rh(t)))
                           = BV_nat(Y) × BV_nat(X)
```

Mais ce théorème ne peut être directement prouvé. Nous devons tout d'abord établir une généralisation : un invariant de boucle. Cet invariant indique que $\forall t \leq size$, le produit de `Y` avec une partie de `X` est égal à la concaténation d'une partie de `Rl` et avec tous les bits de `Rh`. Nous supprimons aussi la condition sur le compteur de boucle :

---

6. Si ces valeurs n'avaient pas été constantes, nous aurions modélisé `X` et `Y` par des fonctions sur le temps.

```
∀ t:nat  t≤size →
       BV_nat(concat(field(Rl(t),size-t,size-1),Rh(t)))
       = BV_nat(Y) × BV_nat(field(X,0,t-1))
```

Cet invariant est prouvé par induction sur `t`. Le case de base (`t=0`) est simplement résolu par réécriture. Le cas général est bien plus complexe ; nous commençons par séparer la preuve en deux, selon la valeur du bit le moins significatif de `Rl`. Pour le cas où ce bit est `true`, nous utilisons le théorème qui montre la correction de `sum` et `carry`. Finalement, la preuve du théorème final est triviale puisqu'il est seulement une instance de l'invariant.

6. Conclusion

Nous avons présenté dans cet article un environnement qui intègre une méthodologie de spécification et différentes techniques de preuve pour la vérification formelle des processeurs. La méthodologie de spécification est basée sur une approche orientée-objet. Les techniques de preuves sont complémentaires : les méthodes les plus simples et automatiques sont utilisées en priorité ; les méthodes qui demandent une intervention humaine ne sont utilisées que si elles sont indispensables.

De nombreux processeurs ont été vérifiés avec notre environnement. En comparaison avec les autres travaux similaires, nos spécifications sont plus concises et lisibles, et nos temps de preuve souvent meilleurs. De plus, nous avons aussi prouvé des types d'instructions qui n'avaient jamais été étudiés auparavant. Nous avons montré qu'une approche coopérative de différentes techniques permet la réalisation de nombreuses vérifications sans intervention de l'utilisateur.

Notre objectif immédiat est l'extension de cet environnement à la vérification des processeurs modernes «pipelinés». Cette vérification devra tenir compte des éventuels conflits de partage des ressources. Nous espérons pouvoir étendre notre bibliothèque orientée-objet dans ce but.

Bibliographie

[ANC 86] ANCEAU F., *The Architecture of Microprocessors*. Addison-Wesley Publishing Company, 1986.

[ARD 95a] ARDITI L. et COLLAVIZZA H., « An Object-Oriented Framework for the Formal Verifi cation of Processors ». In *European Conference on Object-Oriented Programming*, vol. 952 de *LNCS*, August 1995.

[ARD 95b] ARDITI L. et COLLAVIZZA H., « Towards Verifying VHDL Descriptions of Processors ». In *European Design Automation Conference with EuroVHDL*. IEEE Computer Society Press, September 1995.

[ARD 96] ARDITI L., « *BMDs Can Delay the Use of Theorem Proving for Verifying Arithmetic Assembly Instructions ». In *International Conference on Formal Methods in Computer-Aided Design*, LNCS, Palo Alto, CA (USA), November 1996.

[ASH 90] ASHENDEN P. J., *The VHDL Cookbook*. Public Domain, Dept. Computer Science, University of Adelaide, South Australia, first édition, July 1990.

[BEA 94] BEATTY D. L. et BRYANT R. E., « Formally verifying a microprocessor using a simulation methodology ». In 31^{st} *ACM/IEEE Design Automation Conference*, June 1994.

[BER 95] BERN J., MEINEL C. et SLOBODOVÁ A., « Efficient OBDD-based boolean manipulation in CAD beyond current limits ». In 32^{nd} *ACM/IEEE Design Automation Conference*, June 1995.

[BOL 95] BOLIGNANO D., « Vérification des protocoles d'authentification cryptographiques à l'aide de Coq ». In *Journées du GDR Programmation*, Grenoble, France, Nov. 1995.

[BOR 88] BORRIONE D., CAMURATI P., PAILLET J. et PRINETTO P., « A Functional Approach to Formal Hardware Verification: The MTI Experience ». In *IEEE International Conference on Computer Design ICCD'88*, p. 592–595, Port Chester, New-York, October 1988. IEEE Comput. Soc. Press.

[BOY 88] BOYER R. et MOORE J., *A Computational Logic Handbook*. Academic Press, 1988.

[BRY 86] BRYANT R. E., « Graph-Based Algorithms for Boolean Function Manipulation ». *IEEE Transactions on Computers*, vol. C-35, n° 8, p. 677–691, August 1986.

[BRY 94] BRYANT R. E. et CHEN Y.-A., « Verification of Arithmetic Functions with Binary Moment Diagrams ». Rapport technique CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, June 1994.

[BRY 95] BRYANT R. E. et CHEN Y.-A., « Verification of Arithmetic Circuits with Binary Moment Diagrams ». In 32^{nd} *ACM/IEEE Design Automation Conference*, June 1995.

[BUR 91] BURCH J., « Using BDDs to verify multipliers ». In 28^{th} *ACM/IEEE Design Automation Conference*, June 1991.

[BUR 94] BURCH J. et DILL D., « Automatic Verification of Pipelined Microprocessor Control ». In *Computer-Aided Verification*, vol. 818 de *LNCS*, p. 68–80, 1994.

[COH 87] COHN A., « A Proof of Correctness of the Viper Microprocessor: the First Level ». In *VLSI Specification, Verification and Synthesis*. Kluwer Acad. Publ., January 1987.

[COQ 88] COQUAND T. et HUET G., « The Calculus of Constructions ». *Information and Computation*, vol. 76, n° 2/3, 1988.

[COQ 90] COQUAND T. et PAULIN-MOHRING C., « inductively Defined Types ». In *Colog'88*, vol. 417 de *LNCS*, 1990.

[COR 95] CORNES, COURANT, FILLIÂTRE, HUET, MANOURY, PAULIN-MOHRING, MÃNOZ, MURTHY, PARENT, SAÏBI et WERNER, « The Coq Proof Assistant. Reference Manual. Version 5.10 ». Rapport technique, INRIA Rocquencourt - CNRS - ENS Lyon, 1995.

[COU 89] COUDERT O., BERTHET C. et MADRE J.-C., « Verification of Synchronous Sequential Machines Based on Symbolic Execution ». In *Automatic Verification Methods for Finite State Systems*, vol. 407 de *LNCS*, June 1989.

[COU 96] COUPET-GRIMAL S. et JAKUBIEC L., « Coq and Hardware Verification: a Case Study ». In *International Conference on Theorem Proving in Higher Order Logics*, LNCS, 1996. To appear.

[CYR 94] CYRLUK D., RAJAN S., SHANKAR N. et SRIVAS M., « Effective Theorem Proving for Hardware Verification ». In *Theorem Provers in Circuit Design: Theory, Practice and Experience*, September 1994.

[GAL 94] GALLESIO E., « STklos: a Scheme object oriented system dealing with the TK toolkit ». In *Xhibition 94*, San Jose, Jul. 1994. ICS.

[GOR 83] GORDON M., « Proving a Computer Correct using LCF–LSM Hardware Verification System ». Rapport technique 42, Computer Laboratory, The University of Cambridge, September 1983.

[GOR 85] GORDON M., « Why higher-order logic is a good formalism for specifying and verifying hardware ». Rapport technique 77, University of Cambridge, Computer Laboratory, 1985.

[GOR 92] GORDON M. et MELHAM T., *HOL: a proof generating system for higher-order logic*. Cambridge University Press, 1992.

[GUP 92] GUPTA A., « Formal hardware verification methods: a survey ». *Formal Methods in System Design*, vol. 1, n° 2/3, p. 151–238, October 1992.

[Hun 89] HUNT JR. W. A., « Microprocessor Design Verification ». *Journal of Automated Reasoning*, vol. 5, n° 4, December 1989.

[JOY 90] JOYCE J., « *Multi Level Verification of Microprocessor-Based Systems* ». PhD thesis, University of Cambridge, Computer Laboratory, May 1990.

[LAI 92] LAI Y.-T. et SASTRY S., « Edge-valued binary decision diagrams for multi-level hierarchical verification ». In 29^{th} *ACM/IEEE Design Automation Conference*, June 1992.

[MEY 88] MEYER B., *Object-Oriented Software Construction*. Prentice Hall Int., 1988.

[OWR 92] OWRE S., RUSHBY J. et SHANKAR N., « PVS: A prototype verification system ». In 11^{th} *International Conference on Automated Deduction*, vol. 607 de *LNAI*, 1992.

[PAU 95] PAULIN-MOHRING C., « Circuits as streams in Coq: Verification of a sequential multiplier ». In *Basic Research Action "Types"*, 1995.

[PUL 87] PULOU J., RAINARD J. et THOREL P., « Microprocesseur à Test Intégré MTI – Description fonctionnelle et architecture ». Rapport technique NT/CNS/CC/59, CNET, Grenoble, January 1987.

[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. et LORENSEN W., *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[SCH 94] SCHNEIDER K., KUMAR R. et KROPF T., « Automating Verification by Functional Abstraction at the System Level ». In *International Workshop on Higher-Order Logic Theorem Proving and its Applications*, 1994.

[SEK 89] SEKAR R. et SRIVAS M., « Equational techniques ». In BIRTWISTLE G. et SUBRAHMANYAM P., Eds., *Current Trends in Hardware Verification and Automatic Theorem Proving*, p. 173–217, New-York, 1989. Springer-Verlag.

[STA 93] STAVRIDOU V., *Formal Methods in Circuit Design*. N° 37 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[Ste 90] STEELE JR. G. L., *Common Lisp the Language*. Digital Press, second édition, 1990.

[WIN 90] WINDLEY P. J., « *The Formal Verification of Generic Interpreters* ». PhD thesis, University of California, Division of Computer Science, 1990.

[WON 93] WONG W., « Modelling Bit Vectors in HOL: the `word` Library ». In *Higher Order Logic Theorem Proving and Its Applications.*, vol. 780 de *LNCS*, August 1993.

[ZHU 93] ZHU Z., JOYCE J. et SEGER C., « Verification of the Tamarack-3 Microprocessor in a Hybrid Verification Environment ». In *Higher Order Logic Theorem Proving and Its Applications.*, vol. 780 de *LNCS*, August 1993.

II.4.2 An Object-Oriented Framework for the Formal Verification of Processors

Laurent Arditi, Hélène Collavizza. An Object-Oriented Framework for the Formal Verification of Processors. European Conference on Object-Oriented Programming (ECOOP'95). Edited by W. Olthtoff, volume 952 in LNCS, pages 213-234. Aarhus (Denmark). August 1995.

Cet article présente le modèle orienté objet que nous avons utilisé, pour modéliser aussi bien les différents niveaux de description d'un processeur, que pour modéliser le processus de preuve : interprètes et abstractions.

An Object-Oriented Framework for the Formal Verification of Processors

Laurent Arditi and Hélène Collavizza

Université de Nice – Sophia Antipolis
Laboratoire I3S, CNRS-URA 1376
650, Route des Colles. B.P. 145
06903 Sophia Antipolis Cédex. France
email: arditi@unice.fr, helen@essi.fr

Abstract. We propose an object-oriented approach for the formal verification of processors. This approach has been validated on significant applications. It is based on a class hierarchy that provides the basic components to describe processors at any abstraction level, and to specify verifications to execute.
The originality of our method is to combine an object-oriented model (to ensure *genericity*) and a computer algebra verification system (to ensure *efficiency*). Computer experiments with our framework clearly shown three main advantages: processor descriptions are very easy to write down, the core of the verification system is generic so it may be applied without any modification to different processors, and last, the proof times are very short compared with previous approaches.

1 Introduction

In this section, we first provide general motivations for hardware verification, and then outline our approach in general terms.

1.1 Motivations

In order to produce correct circuits, several verification tools such as correct generators of floor-plans or logic simulators are generally applied during the design process. However, it is widely accepted that the existing CAD tools have two major drawbacks:

- they cannot perform exhaustive proofs: this is due to the combinatorial complexity of the problem,
- they are low-level oriented and therefore they cannot deal with abstract specifications.

For these reasons, attention has been paid these last ten years on general formal verification methods (for a survey and further justifications see [10]). In general, formally verifying a circuit consists in proving that its specification is logically equivalent to its implementation, in the sense of a logical equivalence. More explicitly, the verification process is decomposed in three steps:

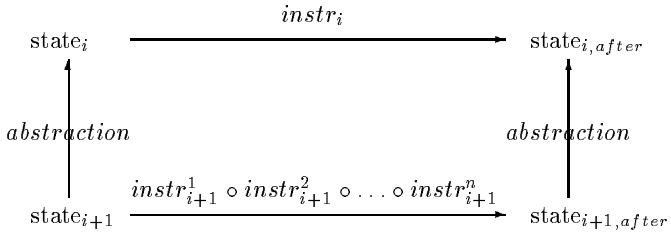
- in the first step, a formal model of the specification of the circuit is defined,
- in the second step, a formal model of the implementation of the circuit is defined, and
- in the third step, a proof that for any input and any state variable values, the two models are equal is performed.

This last point amounts to the proof of a universally quantified formula and can be realized using theorem proving techniques such as simplification or inductive reasoning.

1.2 The Problem of Processor Verification

The previous process is crucial when dealing with complex circuits such as processors. The key to the formal verification is the decomposition in successive proofs at successive abstraction levels: a more and more detailed view of the same processor is described [1]. The *specification* of the circuit is then a description of an abstract level (such as the assembly language level), while its *implementation* is a description at a more concrete level (for instance, the microinstruction level or the electronic block model). For each level, one exactly keeps the same description and verification process. The processor is specified as a set of components on which a set of actions is performed.

Let $level_i$ be an abstract specification level, and $level_{i+1}$ a more concrete one. The set of objects at $level_i$ is included in the set of objects at $level_{i+1}$, and an action at $level_i$ is realized at $level_{i+1}$ by composing a set of actions at $level_{i+1}$. So to verify the correctness between two adjacent levels one must show that the following diagram commutes:



For instance, if $level_i$ is the processor as seen by the assembly programmer, and $level_{i+1}$ is the architecture of the processor, the commutation property of the above diagram means that an assembly instruction is correctly realized by a sequence of microinstructions. Thus, to implement this abstract schema one needs:

- languages to describe states, transitions and abstractions, and
- well-suited proof systems to carry out the transitions and verify the commutation of the diagram.

1.3 Aims of the Paper and Related Works

Several interesting approaches have been proposed to realize formal verification of processors along the lines of the previous general schema, see for instance [7, 11, 18, 21, 22]. However, in our sense, these works display two kinds of shortcomings:

- they are specific to a particular processor. This means that the specification of a new processor must be built from the scratch. As a consequence, they do not provide any user-friendly interface to describe processors and verifications to execute, and
- they are based on very general logical proof tools (such as HOL or Nqthm). However these tools are too complex to support efficiently particular problems.

To overcome the above drawbacks, a new framework for processor verification is presented here. The key idea is to combine a "computer algebra system" and an "object-oriented model" in a single framework. The computer algebra system is used to efficiently simplify expressions. The object-oriented model provides three major advantages:

- the description and verification process is generic, it is the same for all processors and for all specification levels, and
- the object-oriented model is well-suited to the concrete structure of microprocessors. As consequence, the specifications are concise and easy to write down, and
- the object classification and the polymorphism of the methods simplify the core of the verification system.

Computer experiments with our framework have shown that not trivial examples can be treated, in a very efficient way (see the comparative Table 2 at the end of the paper), and with a small specification effort.

Layout of the paper: in section 2, we introduce our method on a simple example. In section 3, we present the basis of our verification framework and emphasize the advantages of our object-oriented approach and of the computer algebra system. In section 4 we deal with the implementation of our framework and in section 5 with the results we have obtained so far.

2 Overview of our Method: an Example

Before discussing the technical details of our method, let us introduce it on a simple example.

Assume we are given a simple processor with a memory and one accumulator register and suppose we wish to prove the "addition" instruction in direct memory addressing mode. Let describe this processor at the assembly language level (level_1) and at the microinstruction level (level_2) as follows:

At the assembly language level, the state of the processor consists of the memory "mem", the accumulator register "acc" and the program counter "pc". The addition instruction in direct memory addressing mode involves the two simultaneous transitions:

ADD_{DIR}:	acc := acc + mem [mem [pc](3 to 7)]	acc *is updated*
	pc := pc + 1	pc *is incremented*

"mem[pc]" contains the code of the addition instruction and, "mem[pc](3 to 7)" extracts the 5 highest bits of "mem[pc]" that contain the operand address.

In our object-oriented framework, this will be described as an instance named "proc-level$_1$" of the class `Interpreter` with attributes `state`, `transitions` and `select` (see Sect. 3.1):

- `state` contains the tuple <mem, pc, acc> where each component is built up from an object-oriented library: "mem" is an instance of the `Memory` class while "pc" and "acc" are instances of the `Register` class.
- `transitions` describes the semantics of each instruction. Here, this attribute includes the set of parallel transitions associated with ADD_{DIR} key.
- `select` selects the current instruction that is in memory at address "pc" (here it selects the addition instruction).

At the microinstruction level, a more concrete state is considered. It includes the assembly language level state plus the instruction register "ir", the operand register "rop" and the current microinstruction pointer "mpc". The addition instruction is executed by a sequence of microinstructions M_0, M_1 and M_2, that involves the following transitions:

M_0 :	ir := mem[pc]	ir *receives the instruction code*
	pc := pc + 1	pc *is incremented*
	mpc := 1	M_1 *is the next microinstruction to execute*
M_1 :	rop := ir(3 to 7)	rop *receives the operand address*
	mpc := 2	M_2 *is the next microinstruction to execute*
M_2 :	acc := acc + mem[rop]	acc *is updated*
	mpc := 0	M_0 *is the next microinstruction to execute*

In our framework, the state and transitions at the microinstruction level will be also described as an instance "proc-level$_2$" of class `Interpreter` as follows:

- `state` contains the tuple <mem, pc, acc, ir, rop, mpc>,
- `transitions` includes the definition of the semantics of M_0, M_1, M_2 as described below,
- `select` selects the microinstruction that is in the microprogram memory at address "mpc".

We will see in Sect. 3.1 that the use of same structures for different levels induces more genericity in the proof process.

In order to perform the proof, we must show that the sequence M_0, M_1, M_2 correctly realizes the transitions associated with ADD_{DIR}. The proof is also specified by an instance of the class `Proof` that has four attributes (see Sect. 3.1):

- `specification` points the object "proc-level$_1$",
- `implementation` points the object "proc-level$_2$",
- `abstraction` defines the state abstraction, that realizes here the hiding of <mem, pc, acc, ir, rop, mpc> into <mem, pc, acc>, and
- `sync` defines the temporal abstraction. Since we assume that the first microinstruction of a microinstruction sequence is always at the address 0 (which is the "fetch" sequence), the predicate `sync` is here "mpc = 0".

To perform this proof, we first take an initial formal value `state` for proc-level$_2$. We then execute the microinstructions selected by `select` of proc-level$_2$ (i.e the microinstruction pointed by "mpc" in the microprogram memory) until `sync` is true. This gives the level$_2$ final state. We also abstract the initial state to obtain the level$_1$ initial state using the `abstraction` attribute. We then execute the instruction selected by `select` of proc-level$_1$ (i.e the instruction pointed by "pc"). This gives the level$_1$ final state. Finally, we verify that level$_1$ final state is an `abstraction` of level$_2$ final state.

In this simple case, the proof does not require any simplification and the state values obtained at the two levels are exactly the same. However, for more complex processors, where addressing mechanisms are much more elaborated, this can involve reducing expressions on bit vectors.

The above specification and proof process will be the same for any processor at any abstraction level.

3 The Framework Basis

In this section, we give further justifications for the object-oriented approach and the use of a computer algebra system.

3.1 Why an Object-Oriented Approach?

The first question that naturally arises is why we choose an object-oriented approach. There are three reasons for that: first, the need for a well-suited model, second, the need for typed processor components and third, the need for genericity. We will illustrate through the next sections of this paper, that the main advantages of object-oriented programming [13, 16] answer to these requirements.

A Well-Suited Model for Processor Description. When teaching computer structure, we usually introduce a processor as a set of components (the data path) that interact according to the signals sent by the sequencer (the control part).

This naturally leads to describe a processor as an object with three attributes:

- `state`: the set of all components visible at the current level,

- `transitions`: a set of state transitions that defines the behaviour of the processor. Each transition is a set of assignments labeled by an instruction key.
- `select`: a function from the processor state that returns the key of the current transition selected by the control part.

The attribute `state` is a set of components that are built out from an object-oriented component library. Thus, its construction takes full benefit of multiple inheritance (see Sect. 4.1).

Processor Components are Strongly Typed Objects. All circuits are built out from the same basic components namely, gates, registers, buses, memories... Each component is characterized by two aspects: its data type and its temporal behaviour and has at least two associated actions: *read* and *write*. Furthermore, some components are specializations of some others. For example, a wire is an instance of a bus (bus of length one), a ROM is an instance of a memory (for which the write action is forbidden). This induces a natural classification of the components that is easily realized with an object-oriented implementation. We will also see that the polymorphism of read and write actions simplifies the core of our verification system.

Specification and Verification Process is Generic. One important point in formally verifying processors, is that the specification and verification process must be reusable for all the processors of a same class. In fact, we need two kinds of genericity:

- *horizontal* genericity: the model should be reusable to specify different processors, and
- *vertical* genericity: at each abstraction level, the same generic model and proof method should be reused.

This leads us to define any processor at any level as an object of the class `Interpreter` that has the three previous attributes `state`, `transitions` and `select`. This ensures the genericity of the description.

In order to ensure the genericity of the proof, a verification between two specification levels is an object of the class `Proof`. This class has two attributes pointing the two interpreters and two others describing the abstraction functions from the implementation to the specification:

- `abstraction`: it defines the state abstraction. It is a function from the implementation to the specification state. It is often a state hiding since the specification state is a subset of the implementation state.
- `sync` defines the temporal abstraction. It is a predicate which value is *true* only when the two levels are synchronized which means that the implementation state corresponds with the start of a transition at the abstract level. This predicate is usually defined as a test on the implementation program counter.

To perform the proof we have to execute a transition at $level_i$ and a sequence of transitions at $level_{i+1}$. The implementation is correct if $level_i$ final state is an abstraction of $level_{i+1}$ final state when `sync` is true.

3.2 Why a Computer Algebra Simplification System?

In order to simplify the expressions involved in state transitions, we use a computer algebra system. This system is less powerful than general provers such as HOL or the Boyer & Moore theorem prover. However, it is enough powerful for processor verification, and furthermore, it is much more efficient and easy to use. It is especially well adapted for this specific problem where expressions to be proved are simple but numerous, and for which efficiency and simplicity must surpass power.

Some other proof systems use rewriting techniques for symbolic expressions (see for example [19, 17]). We choose the computer algebra point of view because it is much more efficient and can easily deal with commutative operators. The simplification of an expression like $(op\ e_1\ \ldots\ e_N)$ is driven by the head operator op, so the reduction strategy associated to op is used. The underlying algebra includes data types such as naturals, bit vectors, Booleans,...

For example, here are the steps reducing the expression

$$\texttt{bv-nat}(\texttt{bv-concat}(V_1, \texttt{bv-concat}(V_2, V_3)))$$

which is the conversion from bit vectors to naturals of the concatenation of three bit vectors. First, using associativity of concatenation we get

$$\texttt{bv-nat}(\texttt{bv-concat}(V_1, V_2, V_3))$$

By applying a specific reduction rule for conversion we get

$$(\texttt{bv-nat}(V_1) * 2^{\texttt{length}(V_2)} + \texttt{bv-nat}(V_2)) * 2^{\texttt{length}(V_3)} + \texttt{bv-nat}(V_3)$$

Finally by using distributivity of multiplication, we obtain

$$\texttt{bv-nat}(V_1) * 2^{\texttt{length}(V_2)+\texttt{length}(V_3)} + \texttt{bv-nat}(V_2) * 2^{\texttt{length}(V_3)} + \texttt{bv-nat}(V_3)$$

4 The Framework Implementation

Having answered the reasons underlying our system, we are now ready to exhibit its components. They are related to two different aspects: the specification process and the verification process.

The first implementation of our framework (see Fig. 1) is realized using a functional and object-oriented language: STk [8], a Scheme with an object system very close to CLOS [20, 14]. STk is a good compromise in order to combine object paradigms and the ability to carry out formal operations.

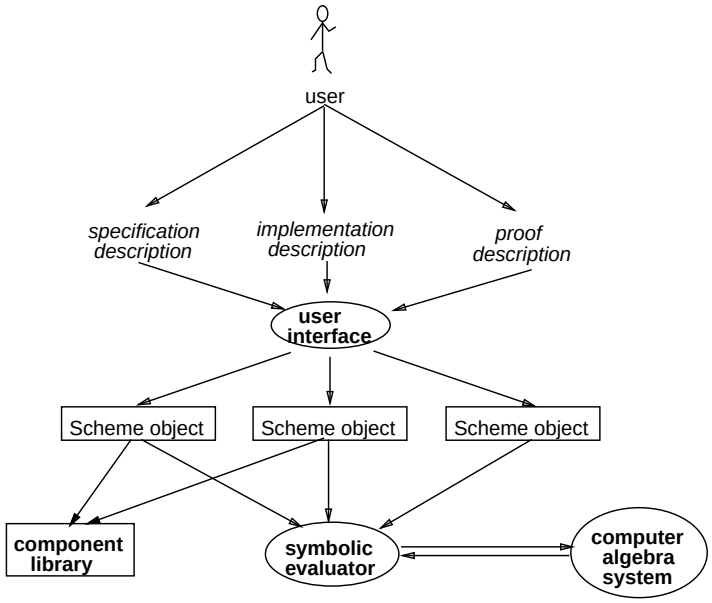


Fig. 1. Overview of the framework

4.1 The Specification Process

We first outline the library from which processor components are built up and then present a specification interface that we have designed in order to facilitate the description process.

A library of components. The attribute `state` of the processor is a set of components built up from a library that consists of a collection of classes describing the variety of structural or behavioural aspects.

Structural Classes: the main data types we use are naturals (`NAT`), bit vectors (`BV`) and arrays of bit vectors (`ARRAY`). Major generic functions are `read` (read contents), `write` (write contents) and `size` (get number of bits). Specific methods are also defined on the different structural classes.

Behavioural Classes: to represent temporal behaviour, we distinguish between what is called in [1] "stored variables" (registers, memories, ...) and "instantaneous variables" (buses or wires). So we have defined the behavioural classes `MEMO` and `NO-MEMO` which are "mixins" [5].

Using multiple inheritance, we get the major component classes to model processors. They have exactly one structural class and one behavioural class as superclasses. Figure 2 shows a simplified class hierarchy.

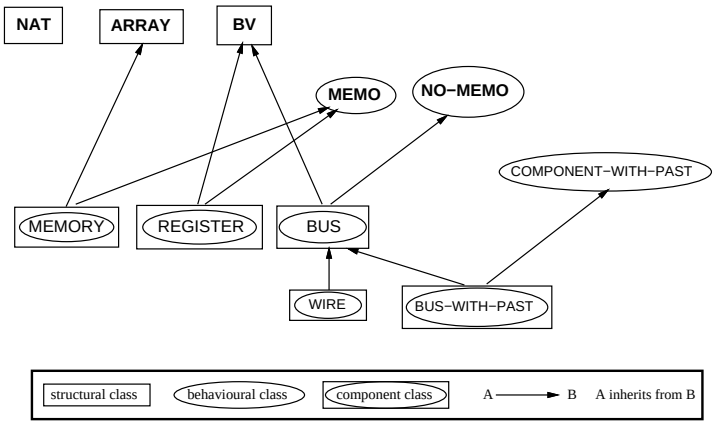


Fig. 2. Hierarchy of component classes

One have to notice that, when building a component class, the behavioural superclass must be more specific than the structural class. In CLOS, this is possible by giving the behavioural class name as the first element of the inheritance list.

Specialized Classes: more specific components such as ROM or stacks, are obtained by specialization of the main classes. Here are examples of class specializations. The first one declares wires as single-bit buses:

```
(define-class Wire (Bus)      ; The wire class inherits from the Bus class.
  ((size :initform 1 :size :getter size)))    ; a wire size is always 1.
```

We define below the `Component-With-Past` class. It allows to keep the history of the component values. This is necessary when modeling complex temporal behaviours such as the communication protocol between the processor and the external memory. The `Component-With-Past` class is also a mixin that does not have any parents. Its `write` method calls for `next-method` that writes the bus value and then memorizes the current value in a stack (`old-values` attribute). This does not produce an error because the class is never instantiated alone: it is *one of the superclasses* of a class, the other superclass is a "component class". The `Component-With-Past` class and its `write` method are defined as follows:

```
(define-class Component-With-Past ()
; The Component-With-Past class is a mixin
   ((old-values :initform '())))              ; The stack of past values
```

```
(define-method write ((c Component-With-Past) value)
; The write method on components with past
  (next-method)                    ; first call on the normal write method
  (slot-set! c 'old-values                  ; and then push the new value
             (push value
                   (slot-ref c 'old-values)))))
```

A component whose past values are required will be built up by inheriting from the class `Component-With-Past` and a component class. For instance, buses with the capacity of getting old values must be declared as a class whose parents are `Component-With-Past` and `Bus`:

```
(define-class Bus-With-Past (Component-With-Past Bus)
; The Bus-With-Past class inherits from
; the mixin Component-With-Past and the Bus class.
  ())
```

Interface Facilities. In order to help system designers that are not familiar with Scheme programming, we have designed a textual interface to describe processors in a proof-oriented style. This interface is a simple language to enforce designers to fill all the fields of interpreter and proof objects. Its syntax is coming closer to the standard of Hardware Description Languages: VHDL. Starting from this more readable form, a translator automatically generates the Scheme internal representation (see Fig. 1).

4.2 The Proof Process

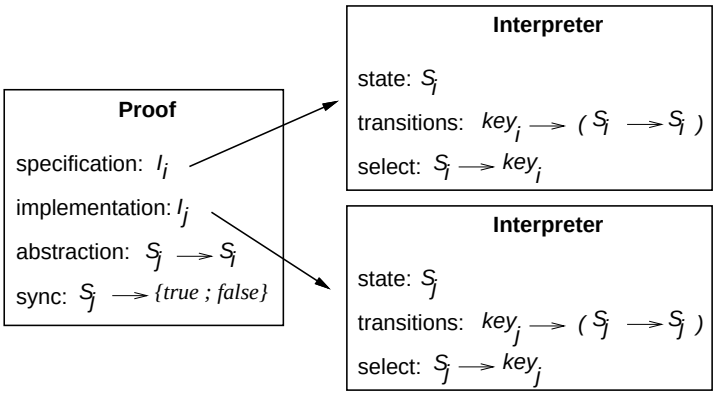


Fig. 3. One proof object and its two interpreters

The Verification Steps. Assume we have defined two objects of the class `Interpreter` that specify a processor at levels $level_i$ and $level_j$, and one object of the class `Proof`, that links these two interpreters (Fig. 3). Then, the correctness proof between the two levels is performed according to the following steps:

1. compute an initial state S_j of $level_j$ such that `sync` is true:

$$S_j \texttt{ := init(}state_j\texttt{)}$$

2. take an abstraction S_i of this state:

$$S_i \texttt{ := abstraction(}S_j\texttt{)}$$

3. execute the transition selected by the $level_i$ interpreter to get the $level_i$ final state

$$\begin{aligned} trans &\texttt{ := transitions(select(}S_i\texttt{))} \\ S_i &\texttt{ := exec-trans(}trans\texttt{, }S_i\texttt{)} \end{aligned}$$

4. execute transitions at $level_j$ until both interpreters are synchronized to get the $level_j$ final state:

```
repeat
   trans := transitions(select(S_j))
   S_j := exec-trans(trans, S_j)
until sync(S_j)=true
```

5. verify that $S_i \equiv$ `abstraction(`S_j`)`

Broadly speaking, points 1 and 2 compute initial states using symbolic values.

Points 3 and 4 perform the state transitions; "`exec-trans`" carries out the transitions while simplifying expressions (its semantics will be detailed next).

Point 5 consists in a syntactic checking of the equality of the two expressions. For most of the processors and assembly instructions the two expressions (that have been first simplified by our computer algebra system) are syntactically equal. Nevertheless, we have also implemented more elaborated proof techniques, such as Binary Moment Diagrams when syntactic verification is not sufficient [2].

A Symbolic Evaluator of Transitions. Let us now describe more precisely how the transitions that define instruction semantics are performed. Assume we are given a transition $dest := source$. Then our symbolic evaluator executes this transition as follows:

- call on `read` method to get the value of *source* if it involves processor components,
- call on the simplification system in order to simplify the resulting value if possible,
- call on the `write` method to modify the value of *dest*.

The symbolic evaluator takes convenience of the object-oriented approach. The evaluation process is generic since `read` and `write` are methods associated to all component classes. Of course, it is not the same thing to read a memory or to read a register but these distinct behaviours are discriminated by inheritance.

A short version of the transition execution method is given below:

```
(define-method exec-trans (dest source (i Interpreter))
; This method executes the assignment of dest with the source value
; over components of the interpreter i
  (let ((rs                  ; the real source value is computed as follows:
          (if (component? source i)   ; if it contains a component of i,
              (simplify (read source))          ; read and simplify.
              (simplify source))))                ; else just simplify
    (write dest rs)))       ; call on the write method on the destination
```

The Verification Kernel. We give here the classes and methods used to implement the proof algorithm. A simplified declaration of the `Interpreter` class is:

```
(define-class Interpreter ()                ; The Interpreter class:
  ((state :init-keyword :state :getter state)
   (select :init-keyword :select :getter select)
   (transitions :init-keyword :transitions :getter transitions)))
```

Some methods are defined for this class in order to reset, update state...

The `Proof` class is as follows:

```
(define-class Proof ()                              ; The Proof class:
  ((s :init-keyword :specification :getter specification)
   (i :init-keyword :implementation :getter implementation)
   (sync :init-keyword :sync :getter sync)
   (abstraction :init-keyword :abstraction :getter abstraction)
   (prove :init-keyword :prove :getter prove)))
```

The `prove` attribute is here to facilitate the proof by cases: it is a set of *all* instructions of the specification interpreter with all their addressing modes.

The method executing the whole proof returns a Boolean showing the correctness. If the proof fails we can get the instruction being proved and the components whose values are wrong.

```
(define-method proof-ok? ((p Proof))
; Returns true if the implementation of p correctly implements its specification
  (mapand (lambda (instr)                ; All instructions are correct?
            (instr-ok? (specification p) (implementation p)
                       instr (abstraction p) (sync p)))
          (prove p)))
```

The `instr-ok?` method discriminates on multiple arguments. It proves or disproves a single instruction: it returns a Boolean showing the equivalence of the two interpreters `s` and `i`, for instruction `instr`, and using the state and temporal abstractions `abstr` and `sync`.

```
(define-method instr-ok? ((s Interpreter) (i Interpreter)
                          instr abstr sync)
; Returns true if s and i are equivalent when executing the instruction instr
; and using abstraction functions abstr and sync.
  (update-state i (new-state i instr))            ; give i a new state
  (update-state s (abstr i))                      ; give s an equivalent new state
  (update-state s (exec (member (select s) (transitions s)) s))
  ; execute the current instruction at the specification level
  (repeat-until
    (update-state i (exec (member (select i) (transitions i)) i))
    ; execute instructions at the implementation level
    (sync i))                                     ; until the temporal abstraction predicate is true
  (equiv? (abstr i) s))                           ; then compare the two final states
```

5 Experimental Results

5.1 Application to Tamarack-3

Tamarack-3 is a benchmark microprocessor described in [9, 12, 19, 22]. Our specification follows the one given in [6] using HOL.

We have specified Tamarack-3 at four levels: the "macro level" which is the assembly level, the "micro level" which is the microinstruction level, the "phase level" which is the decomposition of the "micro level" for each clock phase and the "EBM" which is the structural description of the processor as functional blocks (Fig. 4). We verified the processor in three proof steps between adjacent levels.

At the macro level, components are an accumulator, a register keeping an address to return from subroutines, a program counter, a register latching an external signal of interrupt request, and a register whose value is `1` when an interrupt is being processed. The external memory is also described at this level.

Here is an extract of the specification of Tamarack-3 at the macro level:

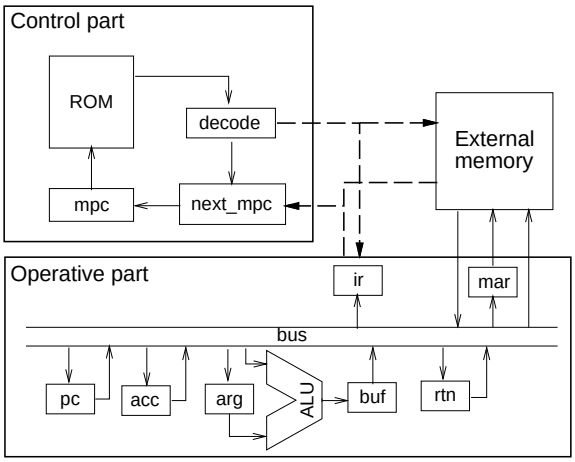


Fig. 4. Structure of the Tamarack-3 microprocessor.

```
(make Interpreter :name 'macro-tamarack
; An instance of the Interpreter class
   :state                              ; The state slot is a list of components
   (list (make Register :name 'pc :size 16)
         ; the program counter is a 16-bit register,
         (make Register :name 'acc :size 16)
         ; as the accumulator
         (make Register :name 'rtn :size 16)
          ; and the return address register,
         (make Memory :name 'mem :size 16)
          ; the external memory is a 16-bit word array,
         (make Register :name 'iack :size 1)
          ; the interrupt acknowledge and
         (make Register :name 'ireq :size 1))
          ; the interrupt request registers are single-bit.
   :select                             ; The select slot is a Lisp expression
   '(if (and ireq (not iack))          ; if an interrupt has to be processed
        ireq-one        ; then return the interrupt pseudo-instruction name
        (decw (fetch mem (address pc))))
                        ; else return the instruction in memory pointed by pc
   :transitions    ; The transitions slot is a list of instruction semantics
   '( (jmp ( (:= pc (fetch mem (address pc))) ))
      ; the jump instruction: assignment to pc
      (adda ( (:= acc (add acc (fetch mem (address pc))))
              (:= pc (inc pc)) ))
      ; the addition instruction: assignment to acc and incrementation of pc
      ...)                                   ; other instructions are here
)
```

At the micro level, the processor components are whose already described at the macro level plus the instruction register, three internal buffer registers, and two registers latching signals between the processor and the external memory. At this level, we also model the microprogram counter.

```
(make Interpreter :name 'micro-tamarack
   :state
   (list (make Register :name 'pc :size 16)
         (make Register :name 'acc :size 16)
         (make Register :name 'rtn :size 16)
         (make Memory :name 'mem :size 16)
         (make Register :name 'iack :size 1)
         (make Register :name 'ireq :size 1)
         (make Register :name 'ir :size 16)   ; the instruction register
         (make Register :name 'arg :size 16)       ; an internal buffer
         (make Register :name 'buf :size 16)            ; another one
         (make Register :name 'mar :size 16)     ; the address register
         (make Register :name 'idle :size 1)   ; for processor/memory
         (make Register :name 'dack :size 1)         ; communication
         (make Register :name 'mpc :size 4))   ; microprogram counter
   :select
   '(case (val4 mpc)              ; the current microinstruction name is
          (0 . u0)            ; u0 when microprogram counter value is 0,
          (1 . u1)            ; u1 when microprogram counter value is 1,
          ... )                                        ; and so on...
   :transitions
   '( (u0                    ; First microinstruction of the fetch sequence:
       ( (:= ir (fetch mem (address pc)))          ; assignment to ir,
         (:= pc (inc pc))                       ; incrementation of pc
         (:= mpc (inc mpc)) ))                                ; and mpc
      (u1 ( ... ))               ; specify u1 and other microinstructions
       ...)                                          ; in the same way.
)
```

The proof description between macro and micro levels is:

```
(make Proof :name 'macro-micro-tamarack
   :specification macro-tamarack              ; points the specification
   :implementation micro-tamarack          ; points the implementation
   :abstraction             ; the state abstraction function is a list where
    ; id is the identity function and nil suppress the corresponding component
   '(id id id id id id nil nil nil nil nil nil nil)
   :sync                   ; the temporal abstraction predicate returns true
   '(= mpc (word4 0))         ; when the microprogram counter value is 0.
)
```

Complete proof. In order to fully prove the Tamarack-3 processor, we have also specified the phase level and the EBM. Our overall goal is to prove that the EBM correctly implements the macro level. In fact we decomposed the proof in three steps: macro versus micro, micro versus phase and phase versus EBM. Transitivity of proofs achieves the complete correctness proof.

5.2 Other Results

We applied our methodology to verify several microprocessors without any change to the `Interpreter` and `Proof` classes, or their methods, nor the proof algorithm of Sect. 4.2. The complexity of these benchmark processors is illustrated in Table 1. The comparative Table 2 shows specification code sizes and proof times for a number of processors already verified with other methods.

Processor	instr.	μinstr.	user registers	word size	addressing modes
AVM-1	30	51	35	32	1
DP-32	20	10	258	32	2
MTI	22	149	38	16	5
Anceau's proc.	8	14	4	16	4
Tamarack-3	8	16	3	16	1

Table 1. Size complexity of processors we specified and proved.

Processor	#SL	SS (p.)	PT (sec.)	other SS	other PT	ref
AVM-1	4	22	1775	110	58 h	[21]
DP-32	2	9	470			
MTI	2	17	1973			
Anceau's proc.	2	4	183			
Tamarack-3	4	3	197	17	10 days	[19]
				?	360 sec.	[22]

Table 2. Processors we specified and proved. #SL is the number of different specification levels, SS is approximative specification code size in number of pages, PT is proof time in seconds on a SUN IPC workstation. Other SS and other PT are specification sizes and proof times of other works referenced in the "ref" column.

Tamarack-3 and Anceau's processors [1] (pages 181–212) are school processors. Their specifications and proofs do not pose any problem. It is interesting to compare results for Tamarack-3. Stavridou [19] used the OBJ3 theorem

prover. It is based on rewriting techniques and one can see that proof time is not acceptable. Furthermore the proof is not automatic. Time proofs in [22] are satisfactory, but to the best of our knowledge, the specifications are given in HOL, which is in our opinion not well-adapted to processor specification.

The other processors are rather complex and their proofs emphasize the usefulness of our specification and proof methodology. DP-32 is a processor described in VHDL in [4]. Its verification has raised one error in its implementation. This proof shows that our framework is able to specify, and then to prove, a processor starting from its VHDL description. A VHDL description style provable in our framework has been derived from its verification [3].

AVM-1 [21] has already been verified by Windley using HOL. Our specification is much more concise than his one, and our proof times are also much more satisfactory.

MTI is a processor designed by CNET in France [15]. It has already been studied but never fully proved. Our system discovered several errors in its design.

6 Conclusion

We have presented an object-oriented approach for processor specification and verification. Any processor at any abstraction level is described by an object of the same interpreter class. A proof between two specification levels is an object of the proof class. The main advantages of this approach is that the verification process is reusable. To verify additional processors, we do not need to modify the verification kernel. Furthermore, the specification process is very easy and systematic since it just consists to instantiate predefined classes. The specifications are very concise and the verification times are very satisfactory compared with other methods.

We are now extending our system in two major directions. First, we are developing a graphical interface for processor description. Thanks to the object framework, this will be a very simple task. Second, we will extend the class of processors we are able to prove, to more complex architectures such as DSP or pipelined architectures. We hope that this extension will only consist to an enrichment of the class hierarchy.

Acknowledgments. We are grateful to J.C. Boussard, E. Kounalis and M. Rueher who had the kindness to comment and improve a preliminary version of this paper.

References

1. F. Anceau. *The Architecture of Microprocessors.* Addison-Wesley Publishing Company, 1986.
2. L. Arditi and H. Collavizza. Binary moment diagrams for verifying loops in microprocessor instructions. Technical report, Laboratoire I3S, Université de Nice – Sophia Antipolis, Nov. 1994.

3. L. Arditi and H. Collavizza. Towards verifying VHDL descriptions of processors. Technical report, Laboratoire I3S, Université de Nice – Sophia Antipolis, Jan. 1995.
4. P. J. Ashenden. *The VHDL Cookbook*. Public Domain, Dept. Computer Science, University of Adelaide, South Australia, first edition, July 1990.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *European Conference on Object Oriented Programming / Object-Oriented Programming Systems, Languages and Applications*, October 1990.
6. M. L. Coe and P. J. Windley. Microprocessor verification: A tutorial. Technical Report LAL-92-10, Laboratory for Applied Logic, Brigham Young University, Provo, Utah, 1992.
7. A. Cohn. A proof of correctness of the Viper microprocessor: the first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Calgary, Canada, Jan. 1987. Kluwer Acad. Publishers.
8. E. Gallesio. STklos: a Scheme object oriented system dealing with the TK toolkit. In *Xhibition 94*, San Jose, Jul. 1994. ICS.
9. M. Gordon. HOL, a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
10. A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, 1(2/3):151–238, Oct. 1992.
11. W. A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, Dec. 1989.
12. J. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Calgary, Canada, Jan. 1987. Kluwer Acad. Publishers.
13. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.
14. A. Paepcke. *Object-oriented programming : the CLOS perspective*. MIT press, 1993.
15. J. Pulou, J. Rainard, and P. Thorel. Microprocesseur à test intégré MTI – description fonctionnelle et architecture. Technical Report NT/CNS/CC/59, CNET, Grenoble, Jan. 1987.
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.
17. R. Sekar and M. Srivas. Equational techniques. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 173–217, New-York, 1989. Springer-Verlag.
18. M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, Sept. 1990.
19. V. Stavridou. *Formal Methods in Circuit Design*. Number 37 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.
20. G. L. Steele Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.
21. P. J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Division of Computer Science, 1990.
22. Z. Zhu, J. Joyce, and C. Seger. Verification of the Tamarack-3 microprocessor in a hybrid verification environment. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications. 6^{th} Int. Work. HUG'93*, volume 780 of *LNCS*, pages 253–266, Vancouver, Canada, Aug. 1993. Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style

Bibliographie

[1] Karlheinz Agsteiner, Dieter Monjau et Soren Schulze : Object-oriented high-level modeling of system components for the generation of VHDL. *In European Design Automation Conference with EURO-VHDL*, Brighton, UK, 1995. IEEE Computer Society Press.

[2] François Anceau : *The Architecture of Microprocessors*. Addison-Wesley Publishing Company, 1986.

[3] Laurent Arditi : *BMDs can delay the use of theorem proving for verifying arithmetic assembly instructions. *In International Conference on Formal Methods in Computer-Aided Design*, volume 1166 de *LNCS*, Palo Alto, CA (USA), novembre 1996.

[4] Laurent Arditi : *Spécification et Preuve des Microprocesseurs*. Thèse de doctorat, Université de Nice Sophia Antipolis, Sophia Antipolis, France, 1996.

[5] Laurent Arditi et Hélène Collavizza : An object-oriented framework for the formal verification of processors. *In European Conference on Object-Oriented Programming*, volume 952 de *LNCS*, août 1995.

[6] Laurent Arditi et Hélène Collavizza : Towards verifying VHDL descriptions of processors. *In European Design Automation Conference with EuroVHDL*. IEEE Computer Society Press, septembre 1995.

[7] Laurent Arditi et Hélène Collavizza : Intégration de techniques coopératives pour la vérification formelle des processeurs. *Technique et Science Informatiques*, 1997.

[8] Peter J. Ashenden : *The VHDL Cookbook*. Public Domain, Dept. Computer Science, University of Adelaide, South Australia, first édition, juillet 1990.

[9] Derek L. Beatty et Randal E. Bryant : Formally verifying a microprocessor using a simulation methodology. *In* 31^{st} *ACM/IEEE Design Automation Conference*, juin 1994.

[10] R.S. Boyer et J.S. Moore : *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[11] Randal E. Bryant et Yirng-An Chen : Verification of arithmetic functions with binary moment diagrams. Rapport technique CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, juin 1994.

[12] Randal E. Bryant et Yirng-An Chen : Verification of arithmetic circuits with binary moment diagrams. *In* 32^{nd} *ACM/IEEE Design Automation Conference*, juin 1995.

[13] Randal E. Bryant et Yirng-An Chen : Verification of arithmetic circuits using binary moment diagrams. *International Journal on Software Tools for Technology Transfer*, 2001.

[14] J.R. Burch et D.L. Dill : Automatic verification of pipelined microprocessor control. *In Computer-Aided Verification*, volume 818 de *LNCS*, pages 68–80, 1994.

[15] Jacques Chazarain et Hélène Collavizza : Combining symbolic evaluation and object oriented approach for verifying processor-like architectures at the RT-level. *In* Milne et Pierre, éditeurs : *Advanced Research Working Conference on Correct HARdware design MEthodologies, CHARME'93*, volume 683 de *LNCS*, pages 109–121, Arles, France, 1993. Springer-Verlag.

[16] E. Clarke, D. Kroening et F. Lerda : A tool for checking ANSI-C programs. *In TACAS 2004*, volume 2988 de *LNCS*, pages 168–176. Springer-Verlag, 2004.

[17] James W. Coffron : *Programmation du 8086 – 8088*. Sybex, 1984.

[18] Avra Cohn : A proof of correctness of the Viper microprocessor : the first level. *In VLSI Specification, Verification and Synthesis*. Kluwer Academic Publisher, janvier 1987.

[19] Th. Coquand et G. Huet : The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[20] Th. Coquand et C. Paulin-Mohring : Inductively defined types. *In Colog'88*, volume 417 de *LNCS*, 1990.

[21] Cornes, Courant, Filliâtre, Huet, Manoury, Paulin-Mohring, Mũnoz, Murthy, Parent, Saïbi et Werner : The Coq Proof Assistant. Reference Manual. Version 5.10. Rapport technique, INRIA Rocquencourt – CNRS - ENS Lyon, 1995.

[22] Solange Coupet-Grimal et Line Jakubiec : Coq and hardware verification : a case study. *In* Joakim von Wright, Jim Grundy et John Harrison, éditeurs : *Theorem Proving in Higher Order Logics, 9th International Conference*, volume 1125 de *LNCS*. Springer, 1996.

[23] Eric Gallesio : STklos : a Scheme object oriented system dealing with the TK toolkit. *In Xhibition 94*, San Jose, Jul. 1994. ICS.

[24] M.J.C Gordon : Why higher-order logic is a good formalism for specifying and verifying hardware. *In* P. A. Subrahmanyam G. J. Milne, éditeur : *Formal Aspects of VLSI Designs*, pages 153–177. Elsevier Science Publisher, 1985.

[25] M.J.C Gordon et T.F. Melham : *HOL : a proof generating system for higher-order logic*. Cambridge University Press, 1992.

[26] Aarti Gupta : Formal hardware verification methods : a survey. *Formal Methods in System Design*, 1(2/3):151–238, octobre 1992.

[27] N.A. Harman et J.V. Tucker : Algebraic models and the correctness of microprocessors. *In* Milne et Pierre, éditeurs : *Advanced Research Working Conference on Correct HARdware design MEthodologies, CHARME'93*, volume 683 de *LNCS*, pages 92–108. IFIP WG 10.2, Springer-Verlag, 1993.

[28] Warren A. Hunt Jr : *FM8501 : a Verified Microprocessor*. Thèse de doctorat, University of Texas at Austin, décembre 1985.

[29] Warren A. Hunt Jr. : Microprocessor design verification. *Journal of Automated Reasoning*, 5(4), décembre 1989.

[30] Jeffrey J. Joyce : Formal verification and implementation of a microprocessor. *In* G. Birtwhistle et P.A. Subrahmanyam, éditeurs : *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, janvier 1987.

[31] M. Kaufmann et J Moore : Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[32] Christoph Kern et Mark R. Greenstreet : Formal verification in hardware design : A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.

[33] D. Kroening, E. Clarke et K. Yorav : Behavioral consistency of C and Verilog programs using bounded model checking. *In 40th DAC*, pages 368–371, 2003.

[34] T.F. Melham : Abstraction mechanisms for hardware verification. *In* G. Birtwistle et P.A. Subrahmanyam, éditeurs : *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[35] Steven P. Miller et Mandayam Srivas : Formal verification of the AAMP5 microprocessor : A case study in the industrial use of formal methods. *In WIFT'95 : Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL (USA), 1995. IEEE Computer Society Press.

[36] K.D. Müller-Glaser et Zippelius R. : An object-oriented extension of VHDL. *In Proceedings of the VHDL-Forum Spring'92 Meeting*, 1992.

[37] Magnus O. Myreen et Michael J.C. Gordon : Verified lisp implementations on arm, x86 and powerpc. *In à paraitre TPHOL*, 2009.

[38] Magnus O. Myreen, Konrad Slind et Michael J.C. Gordon : Extensible proof producing compilation. *In Compiler Construction*, volume 5501 de *LNCS*. Springer-Verlag, 2009.

[39] S. Owre, J.M. Rushby et N. Shankar : PVS : A prototype verification system. *In* 11^{th} *International Conference on Automated Deduction*, volume 607 de *LNAI*, 1992.

[40] J.L. Paillet : A functional model for descriptions and specifications of digital devices. *In* Dominique Borrionne, éditeur : *From HDL Descriptions to Guaranteed Correct Circuits Designs*. IFIP WG10.2, Elsevier Science Publishers B.V (North Holland), 1986.

[41] Christine Paulin-Mohring : Circuits as streams in coq : Verification of a sequential multiplier. Rapport technique, LIP-École Normale Supérieure de Lyon, 1995.

[42] D. Perry : Applying object-oriented techniques to VHDL. *In Proceedings of the VIUF Spring Conference*, pages 217–224, 1992.

[43] Rage3D.com : Intel releases critical core 2 duo microcode update - silently. `http://www.rage3d.com/board/showthread.php?threadid=33889730`.

[44] J. Sawada et Warren A. Hunt Jr. : Verification of FM9801 : An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, March 2002.

[45] Lahiri Shuvendu K., Seshia Sanjit A. et Bryant Randal E. : Modeling and verification of out-of-order microprocessors using UCLID. *In Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 de *LNCS*, pages 142–159. Springer-Verlag, November 2002.

[46] Eric Smith et David Dill : Automatic formal verification of block cipher implementations. *In FMCAD*. IEEE Computer Society, 2008.

[47] Mandayam K. Srivas et Steven P. Miller : Applying formal verification to a commercial microprocessor. *In* Johnson, éditeur : *International Conference on Computer Hardware Description Languages*, Chiba, Japan, 1995.

[48] William Stallings : *Organisation et architecture de l'ordinateur*. Pearson Education France, Informatique, Dept. Computer Science, University of Adelaide, South Australia, 6 ème édition, 2003.

[49] Guy L. Steele Jr. : *Common Lisp the Language*. Digital Press, second édition, 1990.

[50] Miroslav N. Velev et Randal E. Bryant : Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.

[51] Stephen A. Ward et Robert H. Halstead : *Computation Structures*. MIT Press, 1989.

[52] Phillip J. Windley : *The Formal Verification of Generic Interpreters*. Thèse de doctorat, University of California, Division of Computer Science, 1990.

[53] Phillip J. Windley : The practical verification of microprocessor designs. *In* M. Archer, J. Joyce, K.N. Levitt et P.J. Windley, éditeurs : *International Workshop on the HOL Theorem Proving System and its Applications - HOL'91*, pages 32–37, Davis, California, 1991. IEEE Computer Society Press.

[54] Phillip J. Windley : A theory of generic interpreters. *In* Milne et Pierre, éditeurs : *Advanced Research Working Conference on Correct HARdware design MEthodologies, CHARME'93*, volume 683 de *LNCS*, pages 122–134, Arles, France, 1993. Springer-Verlag.

Chapitre III

Vérification formelle des programmes par programmation par contraintes

Ce chapitre présente mes travaux sur la vérification formelle des programmes, travaux accomplis en collaboration avec Michel Rueher puis Pascal Van Hentenryck de l'Université de Brown. Ils ont été menés dans le cadre du projet RNTL "DANOCOPS" (Détection Automatique de NOn-Conformités d'un Programme vis-à-vis de sa Spécification) [34]. Il s'agit de travaux novateurs puisque la vérification formelle des programmes n'avait pas été encore abordée ou très peu avec les contraintes. Ces travaux ont donné lieu au co-encadrement de deux projets de master, celui de Lydie Blanchet pour l'intégration du solveur SAT "SAT4J" et celui de Sébastien Derrien et Eric Le Duff pour la réalisation d'un plug-in eclipse de traduction de Java vers une forme intermédiaire XML, ainsi qu'une interface graphique de visualisation de l'exécution du programme. Je co-encadre actuellement avec Michel Rueher la thèse de Le Vinh Nguyen dans un prolongement de ces travaux qui concernent les systèmes temps réel (voir les perspectives section VI page 259). Notons aussi qu'un gros effort d'implémentation logicielle a été nécessaire ; je l'ai réalisé en majeure partie, notamment le cœur de l'outil de génération du système de contraintes.

Ce chapitre est structuré de la façon suivante. La première section décrit le cadre de nos contributions. La deuxième section présente la programmation par contraintes en domaines finis et les travaux d'Arnaud Gotlieb et Michel Rueher sur le test logiciel qui ont posé les principes de base de la traduction d'un programme en un ensemble de contraintes. Les deux sections suivantes détaillent les deux approches que nous avons proposées : une approche avec génération statique d'un système de contraintes combinant booléens pour la structure de contrôle du programme et entiers pour la partie opérative, et une approche avec génération dynamique du système de contraintes par l'exploration du graphe de flot de contrôle du programme. Je termine par une discussion sur l'état de l'art des méthodes de vérification formelles de programmes.

III.1 Contributions

Comme dans le cas du matériel, le comportement des logiciels est vérifié en pratique en appliquant un ensemble de jeux d'essai, sans passage par une spécification véritable. Un tel procédé est loin d'être toujours suffisant ; ainsi, si la présence d'exemples qui mettent en défaut un comportement attendu du système suffit à remettre en cause sa correction, aucune conclusion ne peut rigoureusement être tirée chaque fois que l'ensemble des jeux de test n'est pas exhaustif. Développer des méthodes pour la *vérification formelle des programmes* constitue donc une problématique d'importance majeure, en particulier pour toutes les applications où le coût d'une erreur de développement, en terme humain ou économique, est très élevé.

De nombreuses méthodes de vérification formelle de logiciel ont été proposées, incluant notamment :

1. le model checking, méthode d'exploration des états atteignables d'un programme, combiné à des techniques d'abstraction ;
2. le bounded model checking, où l'exploration des états est bornée par une certaine profondeur. Ces deux premières méthodes sont également utilisées en

vérification de matériel ; je les ai introduites section I.2.2 page 8,

3. l'*analyse statique abstraite*, basée sur un calcul de point fixe des valeurs du programme ;
4. les méthodes qui procèdent par *raffinements successifs* pour construire des programmes corrects à partir de leur spécification, comme la méthode B [1] ;
5. les *méthode déductives*, où un démonstrateur de théorèmes permet de définir formellement la sémantique du langage et de générer des obligations de preuve.

Notre contribution se situe dans le cadre du bounded model checking, avec utilisation d'une technique d'abstraction. Nous ne parlerons pas de la méthode 4 qui est trop éloignée de notre démarche, puisque notre objectif est de valider des spécifications quelconques, qui peuvent être partielles et affinées a posteriori de l'écriture du programme. Les méthodes de type 5 seront étudiées en partie dans le chapitre VI, section IV page 163 où je présente mes travaux sur la vérification de programmes basée sur la sémantique, avec HOL4. La comparaison des résultats expérimentaux avec les autres approches est présentée section III.3.4 et III.4.5. Une discussion plus générale sur l'état de l'art est donnée section III.5.

III.1.1 Cadre de l'étude

Notre étude concerne la *vérification partielle* de programmes non concurrents représentés par un triplet de Hoare $\{pre\}prog\{post\}$ où *pre* et *post* sont des formules logiques qui représentent la *pré-condition* (i.e. propriété vraie *avant* exécution du programme) et la *post-condition* (i.e. une propriété qui doit être vraie *après* exécution du programme), et *prog* est le programme.

Nous ne traitons pas la détection d'erreur de sûreté comme les débordements dans les calculs ou lors des accès à un tableau (ceci est évoqué dans les perspectives chapitre VI page 259).

Nous nous restreignons à la *vérification incomplète*. En effet, en présence de boucles, nous supposons que le nombre d'itérations est connu ou qu'une borne maximale est fournie par l'utilisateur (e.g. la complexité dans le pire des cas de l'algorithme). Nous effectuons donc un bounded model checking. Nous ne nous préoccupons pas de la terminaison des programmes, mais pouvons au contraire, grâce à la finitude des chemins explorés, trouver certains cas d'erreur dans un programme qui ne termine pas.

Nous nous concentrons principalement sur des exemples de programmes de calcul sur les *entiers et tableaux d'entiers*, contenant des boucles. Nous supposons que la taille des tableaux est fixée, ou à défaut, nous la fixons égale au nombre maximum de dépliages. Nous nous sommes en particulier appuyés sur des exemples de programmes qui implémentent des algorithmes classiques des cours d'algorithmique comme des programmes de tri. En effet, une de mes motivations était de développer un outil que je pourrai utiliser en enseignement lors de travaux dirigés d'algorithmique, afin de mieux convaincre les étudiants de leurs erreurs de programmation. Nous verrons que ces exemples, bien que petits en taille, ne sont pas triviaux à vérifier.

En ce qui concerne le choix du langage, nous avons décidé en collaboration avec nos partenaires du projet DANOCOPS [34], de nous intéresser plus particulièrement

L : *list of instructions*; I : *instructions*; B : *Boolean expressions*
E : *integer expressions*; A : *arrays*; V : *variables*

$L ::= I; L \mid \epsilon$
$I ::= A[E] \leftarrow E \mid V \leftarrow E \mid if\ B\ I \mid while\ B\ I \mid assert(B) \mid enforce(B) \mid return\ E \mid \{L\}$
$B ::= true \mid false \mid E > E \mid E \geq E \mid E = E \mid E \neq E \mid E \leq E \mid E < E$
$B ::= \neg B \mid B \wedge B \mid B \vee B \mid B \Rightarrow B$
$E ::= V \mid A[E] \mid E + E \mid E - E \mid E \times E \mid E/E \mid$

FIG. III.1 – Syntaxe des programmes

à des *programmes écrits en Java dont la spécification est écrite en JML* (Java Modeling Language [33]). En effet, nous souhaitions étendre l'étude réalisée dans le cadre du projet INKA [22] aux langages à objet, et nos partenaires du LIFC avaient déjà réalisé l'outil "BZ-testing tool" de test de programmes Java avec des contraintes [10]. Ainsi, s'est dessiné le choix du langage Java. Les arguments qui nous ont fait retenir JML comme langage de spécification sont les suivants :

- la richesse de son pouvoir d'expression, avec des quantificateurs finis, la notion de comportement normal ou en cas d'exception, la notion d'invariants, de pré- et post-conditions, ... ;
- le fait que JML soit un langage d'annotations qui manipule directement les variables du programme. Ainsi, il n'est pas besoin de mécanisme d'abstraction pour faire le lien entre la spécification et le programme ;
- l'activité de la communauté JML qui propose de nombreux outils de test ou de vérification.

Pour l'instant, nous ne considérons qu'un sous-ensemble très restreint (impératif) de Java présenté figure III.1. D'un point de vue pratique, nos outils prennent en entrée des programmes Java qui suivent cette syntaxe et sont analysés grâce à l'API standard JDT de l'environnement Eclipse [27].

Enfin, notre étude a concerné l'apport des *contraintes en domaines finis*. L'idée principale étant de bénéficier d'un cadre uniforme pour traiter aussi bien du numérique que du booléen, alors que, à l'époque où ces travaux ont débuté, les approches basées sur le model checking ou le bounded model checking nécessitaient une traduction au niveau booléen. Depuis lors, l'émergence des solveurs SMT [24] rend cette observation moins pertinente, mais l'intérêt des contraintes est toujours de mise dans le cas où le programme contient des expressions non-linéaires, que les solveurs SMT ne savent pas traiter actuellement.

III.1.2 Contributions

L'idée générale pour la vérification formelle d'un triplet de Hoare $\{pre\}prog\{post\}$ en utilisant les contraintes sur domaines finis est la suivante :

- *pre*, *prog* et $\neg post$ sont traduits en un ensemble de contraintes C_{pre}, C_{prog} et C_{not_post} selon le principe proposé par Arnaud Gotlieb et Michel Rueher pour la génération de jeux de tests (voir section III.2.2). Le domaine des variables est fixé à $[-2^{f-1}, 2^{f-1} - 1]$ où f est le format des entiers.
- Le système de contraintes $C = C_{pre} \wedge C_{prog} \wedge C_{not\_post}$ est résolu,
- Si C a une solution, alors cette solution est un cas d'erreur d'utilisation

puisqu'elle satisfait les contraintes de la pré-condition et du programme mais viole les contraintes de la post-condition.
- Si C n'a pas de solution, alors $C_{pre} \wedge C_{prog} \wedge C_{not\_post}$ n'a pas de modèle dans $[-2^{f-1}, 2^{f-1} - 1]$ et donc $\neg(C_{pre} \wedge C_{prog} \wedge C_{not\_post}) = (C_{pre} \wedge C_{prog}) \Rightarrow C_{post}$ est valide dans $[-2^{f-1}, 2^{f-1} - 1]$

Il s'agit d'un usage novateur d'un solveur de contraintes qui relève des défis difficiles. Les domaines des variables sont très grands puisqu'il s'agit de tous les entiers représentables en machine. Les applications habituelles des contraintes ont des domaines plus petits, comme par exemple un ensemble de tâches à effectuer pour un problème d'emploi du temps. Le système de contraintes contient de nombreuses contraintes gardées (i.e. conditionnelles, voir section III.2.1) qui proviennent des instructions de contrôle du programme. Les domaines des variables des gardes sont difficiles à réduire quand il s'agit des variables d'entrée du programme, puisque celui-ci doit être correct quelles que soient les valeurs d'entrée. La résolution de ces contraintes gardées est donc peu efficace. Enfin, dans le cas où le programme est correct, le système de contraintes n'a pas de solution. Cela peut donc impliquer le parcours de la totalité de l'arbre de recherche et les stratégies de recherche développées dans la communauté de programmation par contrainte pour trouver efficacement une première solution ont donc un intérêt limité.

Dans ce contexte, nous avons proposé deux approches. La première approche combine contraintes booléennes pour la partie contrôle du programme et contraintes sur les entiers pour la partie opérative. L'idée est similaire à l'abstraction de prédicats dans le model checking symbolique (voir section I.2.2) mais le fait de résoudre un système hybride qui contient à la fois les entiers et les booléens permet de réduire le nombre de contre-exemples fallacieux. La deuxième repose sur une exécution symbolique du programme, en coupant à la volée les chemins qui ne sont pas exécutables. Elle combine un solveur booléen, un solveur linéaire et un solveur en domaines finis. Les points forts de ces travaux se résument ainsi :

- Nous avons montré la *pertinence de la programmation par contraintes pour la validation de programmes.*
- Nous avons mis en évidence les *atouts des résolutions hybrides* qui combinent plusieurs solveurs.
- Nous avons appliqué notre méthodologie sur de *nombreux exemples de taille significative.*
- Les résultats expérimentaux montrent que ces approches se *comparent favorablement* aux autres approches basées sur le bounded model checking.

La section suivante rappelle les principes de base des contraintes en domaines finis et montre comment elles ont été utilisées pour la génération de jeux de test.

III.2 Les contraintes en domaines finis et leur application à la génération de jeux de tests

III.2.1 Les contraintes en domaines finis

La programmation par contraintes [3, 21] est dédiée à la résolution de problèmes de recherche difficiles, tels les problèmes d'emploi du temps, de planification, de routage, ... Ses avantages principaux sont l'efficacité en terme de temps d'exécution, ainsi que le pouvoir d'expression et la facilité d'utilisation pour des applications cibles. En effet, de nombreuses contraintes ont été définies afin de modéliser au mieux un ensemble d'applications spécifiques comme par exemple, la contrainte globale de cardinalité qui exprime que la somme des valeurs d'un tableau doit être égale à une certaine valeur, comprise entre deux bornes. Cette contrainte est utile pour les problèmes d'emploi du temps pour exprimer qu'un employé doit travailler un nombre d'heures par jour compris entre un minimum et un maximum. Des algorithmes particuliers, souvent basés sur des parcours de graphe, permettent de résoudre de telles contraintes de façon efficace (voir l'habilitation à diriger des recherches de Jean-Charles Régin pour une présentation claire et détaillée d'exemples de contraintes globales, avec leurs domaines d'application ainsi que leurs algorithmes de résolution [40]).

Définition III.2.1 (CSP sur domaines finis)
Un problème de satisfaction de contraintes (CSP par la suite pour Constraint Satisfaction Problem*) se caractérise par :*

- *un ensemble de* **variables** $X = \{x_1, ..., x_n\}$,
- *un ensemble de* **domaines** D_{x_i} *des valeurs possibles pour chaque variable* x_i,
- *un ensemble de* **contraintes** $C = \{c_1, ..., c_n\}$ *qui restreignent les valeurs que les variables peuvent prendre simultanément.*

Un CSP est dit à **domaines finis** *quand les* D_{x_i} *sont des ensembles finis comme par exemple les booléens, les entiers bornés, ou un ensemble fini de valeurs écrites en extension (e.g. un ensemble de couleurs)*[1].

Une *solution d'un CSP* est une affectation de chacune de ses variables avec une valeur de son domaine qui satisfait toutes les contraintes.

Dans la suite de ce chapitre, nous noterons $Var(c)$ l'ensemble des variables présentes dans la contrainte c et nous utiliserons indifféremment une notation ensembliste comme dans la définition ci-dessus ou une conjonction pour représenter un ensemble de contraintes. Enfin, nous considérerons uniquement les CSP à domaines finis ; les CSP en domaines continus seront le cœur du chapitre V page 209.

Résolution d'un CSP La recherche d'une solution d'un CSP repose sur les trois mécanismes suivants :

- le *filtrage*, qui élimine du domaine d'une variable les valeurs qui ne satisfont pas une propriété de *consistance* par rapport à une ou plusieurs contraintes,

[1] Les CSP en domaines continus sont définis section V.1.6 page 213.

– la *propagation*, qui propage les informations obtenues sur une des variables x afin de réduire le domaine d'une variable y qui intervient dans une contrainte contenant x (i.e. une variable y telle que $\exists\ c,\ x \in Var(c) \wedge y \in Var(c)$),
– la *recherche*, qui sélectionne une variable et une valeur pour cette variable[2] quand le filtrage et la propagation n'ont pas permis de réduire les domaines à un singleton.

La sélection d'une valeur pour une des variables peut induire une réduction du domaine d'autres variables lors du prochain filtrage. Ces étapes sont donc répétées en séquence jusqu'à ce qu'une solution soit trouvée.

La base de l'étape de filtrage est d'assurer la *consistance d'arc*.

Définition III.2.2 (Consistance d'arc [35, 37])
Une contrainte c est arc-consistante si et seulement si :

$\forall\ x_i \in Var(c), \forall\ v_i \in D_{x_i}, \exists\ v_0 \in D_{x_0}, ..., v_{i-1} \in D_{x_{i-1}}, v_{i+1} \in D_{x_{i+1}}, ..., v_n \in D_{x_n}$ *tels que* $c(v_0, ..., v_n)$ *est vraie.*

En d'autres termes, la contrainte c est arc-consistante si pour chaque variable x_i dans $Var(c)$, chaque valeur dans D_{x_i} a un support dans le domaine des autres variables de $Var(c)$.

Exemple III.1 (Filtrage par arc–consistance) Soit le CSP $X = \{x_1, x_2, x_3\}$, $D_{x_1} = \{3, 4, 5, 6\}, D_{x_2} = \{0, 1, 2, 3, 4, 5\}, D_{x_3} = \{1, 2, 3\}, C = \{x_1 < x_2, x_3 = x_2 - 2\}$. Une étape de filtrage par consistance d'arc pour la contrainte $x_1 < x_2$ va éliminer de D_{x2} les valeurs 0, 1, 2 et 3 pour obtenir $D_{x2} = \{4, 5\}$. Ce nouveau domaine est propagé pour filtrer le domaine de x_3 en utilisant la contrainte $x_3 = x_2 - 2$; on obtient $D_{x_3} = \{2, 3\}$. Le filtrage et la propagation ne permettent plus de réduire encore le domaine des variables. Intervient alors l'étape de recherche. On choisit une première valeur pour x_1, par exemple la valeur 3. Cela ne permet pas de réduire les autres domaines. On énumère donc également sur x_2 en choisissant la valeur 4 ce qui donne la première solution : $(3, 4, 1)$. ♯

Notons que la consistance d'arc est une consistance *locale* qui réduit le domaine des variables en considérant une seule contrainte à la fois. Par conséquent, un système peut être arc-consistant et ne pas avoir de solution, comme le CSP suivant : $X = \{x_1, x_2, x_3\}, C = \{c_1 : x_1 \neq x_2, c_2 : x_2 \neq x_3, c_1 : x_3 \neq x_2\}, D_{x_1} = D_{x_2} = D_{x_3} = \{0, 1\}$. C'est l'étape d'énumération qui permettra de réduire tous les domaines de ce CSP à l'ensemble vide puisque ce CSP n'a pas de solution. Un tel système sera par contre résolu très efficacement par une consistance *globale* en utilisant la contrainte "all-diff", qui travaille sur plusieurs contraintes à la fois [40].

La figure III.2 présente le schéma standard d'un algorithme de filtrage par consistance d'arc AC3 [34], où C est l'ensemble des contraintes, D est le produit cartésien des domaines D_{x_i}, `filtrage_AC` est l'opération de filtrage par arc–consistance. Nous verrons au chapitre V page 209 que ce schéma est également utilisé pour les domaines continus ; dans ce cas, la fonction de filtrage `filtrage_AC` est remplacée par une relaxation de l'arc–consistance qui utilise les bornes du domaine.

[2] en utilisant des heuristiques efficaces comme par exemple, la variable qui est la plus contrainte (i.e. intervient dans le plus grand nombre de contraintes), ou qui a le plus petit domaine.

```
AC3( C, D)
    queue ← C
    while queue ≠ ∅ {
        c ← enlever(queue)
        D' ← filtrage_AC(c, D)
        if D' ≠ D {
            D ← D'
            queue ← queue ∪ {c' ∈ C | Var(c) ∩ Var(c') ≠ ∅}
        }
    }
```

FIG. III.2 – Algorithme générique de filtrage par arc–consistance AC3

Contraintes gardées Les contraintes gardées sont des contraintes conditionnelles dont l'évaluation dépend d'autres contraintes. Nous utilisons ces contraintes dans notre première approche pour traduire les instructions conditionnelles des programmes (voir section III.3 page 99). Soit $C_0 \rightarrow C_1$ une contrainte gardée où C_0 et C_1 sont des conjonctions de contraintes basiques. La relation $C_0 \rightarrow C_1$ exprime que les contraintes C_1 doivent être ajoutées au système de contraintes courant quand le solveur peut prouver que la conjonction C_0 est satisfaite. Plus précisément :

- quand le solveur peut prouver que C_0 est satisfaite, alors les contraintes C_1 sont ajoutées à l'ensemble des contraintes ;
- quand le solveur peut prouver que C_0 n'est pas satisfaite, alors la contrainte gardée $C_0 \rightarrow C_1$ est effacée ;
- tant que le solveur ne peut prouver ni que C_0 est satisfaite, ni qu'elle n'est pas satisfaite (i.e. il n'a pas assez d'information sur le domaine des variables de C_0), alors la contrainte gardée n'est pas considérée, on dit alors qu'elle est "gelée".

Notons que la plupart des solveurs utilisent une quatrième règle de *propagation inverse* afin d'accroître l'efficacité :

- quand le solveur peut prouver que C_1 n'est pas satisfaite, alors les contraintes $\neg C_0$ sont ajoutées à l'ensemble des contraintes

Le mécanisme de propagation fait que le solveur essaie de prouver la garde C_0 d'une contrainte gelée à chaque fois que le domaine d'une de ses variables a été réduit. Mais tant que la garde ne peut pas être résolue, la contrainte $C1$ n'est pas propagée. Cela induit un problème d'efficacité lorsque les gardes portent sur des variables dont le domaine est grand, comme c'est le cas pour la vérification des programmes (voir section III.3.5).

III.2.2 Génération de jeux de test par programmation par contraintes

Nos travaux sur la vérification formelle des programmes reprennent en partie les mécanismes de génération d'un CSP à partir d'un programme qui ont été définis dans la thèse d'Arnaud Gotlieb pour le test logiciel [28]. Ces travaux ont été réalisés en partie dans le cadre du projet RNTL INKA [22], dont le thème était la génération automatique de cas de test structurel pour des programmes écrits en C/C++. Nous verrons que le point critique de cette traduction concerne les instructions conditionnelles, pour lesquelles nous avons proposé plusieurs mécanismes de traduction (voir III.3 et III.4).

Un des objectifs du projet INKA était de générer des données d'entrée pour lesquelles un point sélectionné dans une procédure est exécuté. Les principes mis en œuvre sont les suivants [29, 30, 9] :

- La procédure est transformée statiquement en un système de contraintes en utilisant les techniques statiques d'assignation unique et les dépendances de contrôle du programme.
- Le point à atteindre dans la procédure est transformé en une contrainte ; cette contrainte correspond au chemin à suivre dans le graphe de flot de contrôle pour atteindre ce point.
- La résolution du système de contraintes obtenu permet de vérifier s'il existe au moins un chemin exécutable passant par le point choisi et de produire des cas qui correspondent à un ou plusieurs de ces chemins.

Forme SSA Le point de départ de la traduction en CSP est la forme statique d'affectation unique, SSA pour "Single Static Assignment" [20]. Dans cette forme, chaque variable a une définition unique et chaque utilisation peut être atteinte à partir de cette définition. Pour une instruction d'affectation simple, la forme SSA est simplement un renommage. Pour les instructions conditionnelles, les Φ-fonctions permettent de joindre les définitions d'une même variable faites dans des branches différentes. Soit par exemple le programme de la figure III.3 qui calcule la somme des n premiers entiers bornée par 100 (i.e. si $n > 100$, on calcule la somme jusqu'à 100 seulement). La Φ-fonction $max_3 = \Phi(max_1, max_2)$ détermine la valeur de la variable max_3 en fonction du chemin pris dans l'instruction conditionnelle qui précède : max_3 vaut max_1 si elle a été redéfinie dans la partie if et vaut max_2 si elle a été redéfinie dans la partie *else*. En ce qui concerne l'instruction *while*, la Φ-fonction $i_2 = \Phi(i_0, i_1)$ détermine la valeur de i_2 : elle vaut i_0 en venant de l'instruction qui précède le while (instruction 3) et vaut i_1 en venant du while. La valeur de i_2 peut donc être calculée de façon dynamique selon le chemin d'où l'on vient.

Génération du système de contraintes Le système de contraintes généré est obtenu quasi directement à partir de la forme SSA grâce à l'utilisation de contraintes gardées. Les instructions élémentaires sont transformées en contraintes, en gérant simplement le renommage. Ainsi, l'affectation successive d'une même variable est traduite par la conjonction de deux contraintes qui se réfèrent à ses renommages successifs. Par exemple, les affectations $x = 0;\ x = x + 1$ sont traduites par les deux

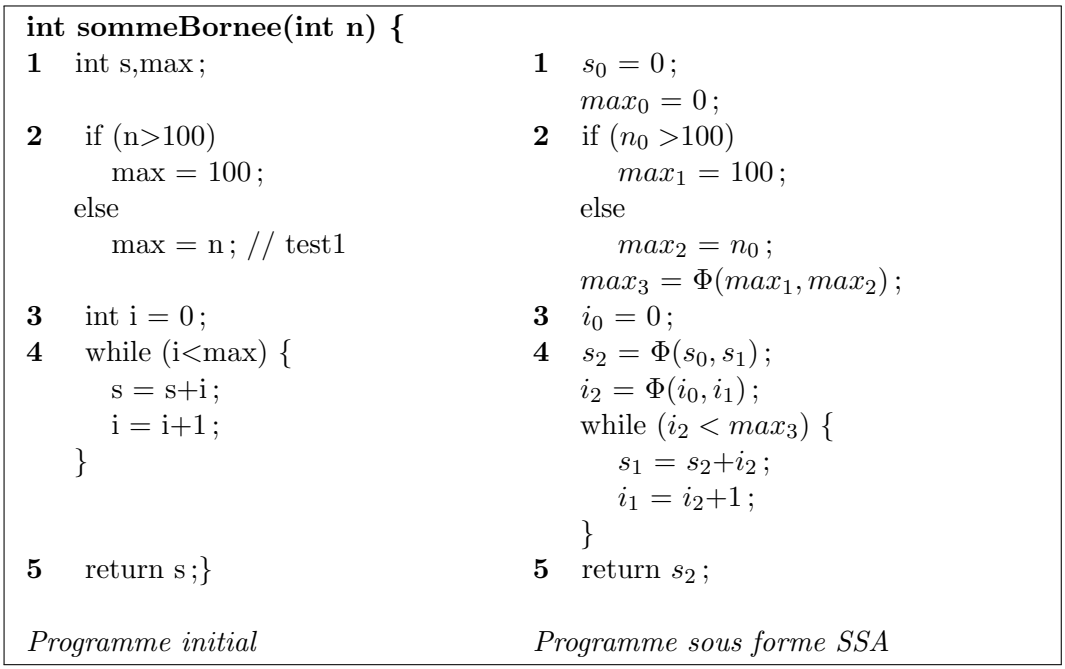


FIG. III.3 – Exemple de forme SSA

contraintes $x_0 = 0 \wedge x_1 = x_0 + 1$. Notons que toutes les opérations élémentaires des programmes contenant des entiers et tableaux d'entiers ont leur traduction sous forme de contraintes. Nous utilisons pour cela les opérateurs booléens et entier usuels, plus la contrainte *element* qui permet d'établir une relation entre un tableau, un indice et une valeur [42]. $element(a, exp_1, exp_2)$ exprime que l'élément d'indice exp_1 du tableau a est égal à exp_2. Dans cette contrainte, la variable a et les expressions exp_1 et exp_2 peuvent être inconnues. Cette contrainte est donc propagée chaque fois que le domaine d'une des variables dans a, exp_1 ou exp_2 a changé.

Une conditionnelle $if\ (c)\ I_1\ else\ I_2$ est traduite par l'opérateur $ite(c, C_1, C_2)$ où C_1 et C_2 sont la traduction en une conjonction de contraintes des blocs d'instructions I_1 et I_2. La sémantique opérationnelle de $ite(c, C_1, C_2)$ est définie par les quatre contraintes gardées suivantes [30] :

1. $c \rightarrow C_1$
2. $\neg c \rightarrow C_2$
3. $\neg(c \wedge C_1) \rightarrow \neg c \wedge C_2$
4. $\neg(\neg c \wedge C_2) \rightarrow c \wedge C_1$

Les deux premières contraintes découlent directement de la sémantique opérationnelle de l'instruction conditionnelle $if\ else$. Les deux dernières contraintes sont redondantes et permettent d'accroître l'efficacité de la résolution des contraintes gardées.

Un opérateur récursif w est défini de façon similaire pour les boucles while. Il utilise également des contraintes gardées en gérant les renommages et en posant dynamiquement les contraintes du corps de la boucle quand la condition d'entrée est vérifiée.

Enfin, le point de test à atteindre dans le programme est associé à une contrainte calculée de façon statique par analyse du flot de contrôle du programme, quand il s'agit d'atteindre un point dans une conditionnelle. Par exemple, s'il s'agit d'atteindre le point de test $test1$ dans le programme de la figure III.3, il suffit d'ajouter la contrainte $\neg(n_0 > 100)$. Quand il s'agit d'atteindre un point à l'intérieur d'une instruction de boucle, la condition pour atteindre ce point est calculée dynamiquement, lors de la pose des contraintes gardées.

Résolution des contraintes gardées Les contraintes gardées générées par les opérateurs *ite* et w sont résolues par un mécanisme de détection d'implication de la condition, qui repose sur une étape de filtrage. Plus précisément, soit $C_1 \rightarrow C_2$ une contrainte gardée et soit σ l'état actuel du CSP (i.e. contraintes et domaines).

- Si le filtrage de $\sigma \wedge \neg C_1$ détecte une inconsistance, alors $\sigma \Rightarrow C_1$. On ajoute donc C_2 au système de contraintes et on enlève la contrainte gardée $C_1 \rightarrow C_2$.
- Si le filtrage de $\sigma \wedge C_1$ détecte une inconsistance, alors $\sigma \Rightarrow \neg C_1$. On enlève donc la contrainte gardée $C_1 \rightarrow C_2$.
- Enfin, si le filtrage ne permet d'établir ni l'une ni l'autre des implications précédentes, alors la contrainte gardée $C_1 \rightarrow C_2$ est gelée.

L'observation essentielle est que, dans le cas du test, les contraintes qui définissent le point à atteindre dans le programme permettent de réduire efficacement les domaines avec un filtrage simple. Nous verrons que ceci n'est plus vrai dans le cas de la vérification formelle des programmes, puisqu'il s'agit ici de tester le programme quelles que soient ses valeurs d'entrée, et donc pour tous les chemins exécutables. Dès lors, quand les conditions portent sur les variables d'entrée du programme, il n'y a que peu d'espoir que le filtrage puisse réduire le domaine de ces variables ; la condition de la contrainte gardée reste alors indéterminée. Nous verrons que nous avons proposé deux approches pour résoudre ce problème : abstraire les conditions par des booléens (section III.3) et effectuer une exécution symbolique à la volée (section III.4). Les sections suivantes présentent ces deux approches.

III.3 Approche par abstraction booléenne des conditions

Notre première approche est basée sur une transformation du programme et de sa spécification en un système de contraintes gardées, selon les principes établis pour la génération automatique de cas de tests (voir III.2.2). Ces travaux ont été présentés à TACAS'06 [17] qui est inclus comme article support section III.6.1 page 127. Nous allons voir que l'utilisation de contraintes gardées pour modéliser les instructions de contrôle du programme empêche la résolution du système de contraintes dans un temps acceptable, et que nous avons introduit des booléens pour abstraire les conditions, en particulier quand il s'agit d'expressions portant sur des variables d'entrées du programme (voir III.3.2).

III.3.1 Construction du système de contraintes

La traduction du programme en un système de contraintes s'inspire des principes présentés en III.2.2. Cependant, nous avons procédé de façon différente dans le cas des instructions de contrôle :

- Pour les instructions conditionnelles, nous générons des contraintes gardées, mais ajoutons également des contraintes de renommage qui jouent le rôle des Φ-fonction pour connaître le renommage des variables après la conditionnelle.
- Pour les instructions de boucle, nous supposons qu'une borne maximale est fournie ou déterminée par les données. Nous déplions les boucles en fonction de cette borne maximale sous la forme de conditionnelles emboîtées.

Dans la suite, nous expliquons comment générer le système de contraintes en décrivant la gestion des renommages et en indiquant pour chaque type d'instruction quelles sont les contraintes qui sont générées. Il s'agit d'une définition inductive : les cas de base sont les expressions et les instructions d'affectation (scalaire et dans un tableau) et le cas général concerne les instructions de contrôle (conditionnelles et boucles).

Fonction de renommage

- σ est la fonction de renommage des variables. C'est une fonction de Var dans $\mathbb{N}$ où Var est l'ensemble des variables (scalaires ou tableaux) du programme. $\sigma(x)$ est le numéro de renommage courant de la variable x.
- $next(\sigma, x)$ représente la fonction de renommage qui est partout égale à σ sauf pour le renommage associé à x qui est égal à $\sigma(x) + 1$.
- $x_{\sigma(x)}$ est la variable du CSP associée au renommage courant de x pour la fonction de renommage σ. Par exemple, si le renommage courant de x dans σ est 3, c'est-à-dire $\sigma(x) = 3$, alors la variable associée est notée x_3.

Variables Chaque déclaration de variable x et chaque paramètre p de la procédure sont associés à une variable x_0 (ou p_0) du CSP. Les variables scalaires ont pour domaine $[-2^{f-1}, 2^{f-1} - 1]$ où f est le format des entiers (i.e. f=8,16, 32 ou 64 bits). Nous supposons que les longueurs des tableaux sont connues. Si ce n'est pas le cas, nous les fixons à la profondeur maximale des chemins que nous explorons. Pour chaque variable de type tableau, nous associons donc une variable de type tableau statique dans le CSP.

Expressions Une expression exp du programme est transformée en une expression du CSP en associant à chaque variable x du programme la variable $x_{\sigma(x)}$ du CSP et en associant à chaque opérateur booléen, entier ou d'accès à un tableau l'opérateur correspondant dans le CSP. Nous notons $ssa_\sigma(exp)$ l'expression du CSP qui correspond à l'expression exp du programme. Par exemple, si $\sigma(x) = 1$ et $\sigma(a) = 2$ où a est un tableau, $ssa_\sigma(x + 1) = x_1 + 1$ et $ssa_\sigma(a[x + 1]) = a_2[x_1 + 1]$.

Affectation scalaire Soit $n = \sigma(x)$ le renommage courant de x. L'affectation $x = exp$ est traduite en la contrainte $x_{n+1} = ssa_\sigma(exp)$ et la fonction de renommage σ devient $next(\sigma, x)$.

Affectation dans un tableau Soit $n = \sigma(a)$ le renommage courant de a. L'affectation $a[exp_1] = exp_2$ est traduite en deux contraintes :

1. $element(a_{n+1}, ssa_\sigma(exp_1), ssa_\sigma(exp_2))$
2. $\bigwedge_{j=0}^{length-1}(ssa_\sigma(exp_1) \neq j \rightarrow element(a_{n+1}, j, a_n[j]))$ où $length$ est la longueur du tableau

et la fonction de renommage σ devient $next(\sigma, a)$.

La première contrainte exprime que dans le nouveau tableau a_{n+1}, la case d'indice exp_1 a changé et a pris la valeur exp_2. La conjonction de contraintes gardées exprime que dans le nouveau tableau a_{n+1}, toute case j différente de exp_1 est restée inchangée, et elle est donc égale à $a_n[j]$. Notons que le renommage des tableaux est tout aussi indispensable que le renommage des variables scalaires. Sans ce renommage, les affectations successives $a[i] = 0; a[i] = 3$ seraient traduites par les deux contraintes $element(a, i, 0) \wedge element(a, i, 3)$ ce qui entraînerait la détection d'une inconsistance lors du filtrage associé à la contrainte $element$ puisque $0 \neq 3$.

Exemple III.2 (Affectation dans un tableau) Soit l'affectation dans un tableau $a[i+1] = 3x+2$. On suppose que $\sigma(a) = 0, \sigma(i) = 3, \sigma(x) = 2$ et $a.length = 4$. Cette affectation est traduite en les contraintes suivantes :

$element(a_1, i_3 + 1, 3x_2 + 2)$
$i_3 + 1 \neq 0 \rightarrow element(a_1, 0, a_0[0]) \wedge \ i_3 + 1 \neq 1 \rightarrow element(a_1, 1, a_0[1]) \wedge$
$i_3 + 1 \neq 2 \rightarrow element(a_1, 2, a_0[2]) \wedge \ i_3 + 1 \neq 3 \rightarrow element(a_1, 3, a_0[3])$
La fonction de renommage devient $next(\sigma, a)$. ♯

Instruction conditionnelle $if\ (c)\ b$ Soit b un bloc d'instructions, soit $const_b$ les contraintes générées pour les instructions du bloc b, soit σ la fonction de renommage avant exécution de b, soit σ' la fonction de renommage après exécution de b et soit var_{def} les variables qui sont définies dans b. L'instruction conditionnelle $if\ (c)\ b$ est traduite en l'ensemble de contraintes gardées suivant :

1. $ssa_\sigma(c) \ \rightarrow \ const_b$
2. $\bigwedge_{x \in var_{def}}(ssa_\sigma(\neg c) \ \rightarrow \ \bigwedge_{i=\sigma(x)}^{\sigma'(x)-1} x_{i+1} = x_i)$

La fonction de renommage après exécution de l'instruction conditionnelle $if\ (c)\ b$ est σ'.

La première contrainte exprime que, si la condition est vérifiée, alors les contraintes associées au bloc b doivent être ajoutées. La deuxième sert à donner une valeur aux variables qui sont définies dans le if même quand celui-ci n'est pas exécuté. Elle remplace donc les Φ-fonctions (voir III.2.2) en forçant le même nombre de renommages que le if soit pris ou non.

Exemple III.3 (Conditionnelle) Soit la conditionnelle $if(c < 3)\{x = x+1;\ x = 2 * x;\}$. Si $\sigma(c) = 1$ et $\sigma(x) = 3$ avant exécution de cette conditionnelle, alors on ajoute les contraintes gardées

$(c_1 < 3) \rightarrow \ (x_4 = x_3 + 1 \wedge x_5 = 2 * x_4)$
$\neg(c_1 < 3) \rightarrow \ x_5 = x_4 \wedge x_4 = x_3$
et la fonction de renommage est égale à σ partout sauf en x où elle est égale à 5. ♯

1	if (i < j) x = 0 ;	(i_0<j_0) → x_1=0
	else {	
2	if (i < 30) {	(¬(i_0<j_0)∧(i_0<30))→(x_1=x_0+1∧x_2=x_1+y_0)
	x = x+1 ;	
	x = x+y ;	
	}	
	else {	
3	if (j > 43) x=2 ;	(¬(i_0<j_0)∧ ¬(i_0<30)∧(j_0>43))→ x_1=2
	else x=3 ;	(¬(i_0<j_0)∧ ¬(i_0<30)∧ ¬(j_0>43))→ x_1=3
	}	
	}	
		gestion des renommages pour le if n°1
		(i_0<j_0) → x_2=x_1
		gestion des renommages pour le if n°2
		(¬(i_0<j_0)∧¬(i_0<30)) → x_2=x_1

FIG. III.4 – Exemple d'un `if else` emboîté. Avant exécution du if, $\sigma(i) = \sigma(j) = \sigma(y) = \sigma(x) = 0$. Après exécution du if, $\sigma(x) = 2$ et $\sigma(i) = \sigma(j) = \sigma(y) = 0$.

Instruction conditionnelle $if\ (c)\ b_1\ else\ b_2$ Soient b_1 et b_2 des blocs d'instructions, soient $const_{b1}$ (resp. $const_{b2}$) les contraintes générées pour les instructions du bloc b_1 (resp. b_2), soit σ la fonction de renommage avant exécution de l'instruction conditionnelle, soit σ_1 (resp. σ_2) la fonction de renommage après exécution de b_1 (resp. b_2), et soit var_{def} les variables qui sont définies dans b_1 et b_2. L'instruction conditionnelle $if\ (c)\ b_1\ else\ b_2$ est traduite par l'ensemble de contraintes gardées suivant :

1. $ssa_\sigma(c) \rightarrow const_{b1}$
2. $ssa_\sigma(\neg c) \rightarrow const_{b2}$
3. $\bigwedge_{x \in var_{def}} (ssa_\sigma(\neg c) \rightarrow \bigwedge_{i=\sigma_{min}(x)}^{\sigma_{max}(x)-1} x_{i+1} = x_i)$
   où $\sigma_{max}(x) = max(\sigma_1(x), \sigma_2(x))$ et $\sigma_{min}(x) = min(\sigma_1(x), \sigma_2(x))$.

La fonction de renommage σ est modifiée de la façon suivante : elle est inchangée pour les variables qui ne sont pas dans var_{def} et pour tout $x \in var_{def}$ elle associe à x la valeur $\sigma_{max}(x)$.

Ici aussi, les contraintes ajoutées pour les variables qui ont été définies dans l'instruction conditionnelle (i.e. contraintes ajoutées en 3) servent à obtenir le même nombre de renommages que le if soit pris ou non et remplacent les Φ-fonctions. En effet, ces contraintes imposent l'égalité entre les renommages successifs d'une même variable, du plus petit ou plus grand, quand ce nombre n'est pas le même dans les deux branches. La figure III.4 donne un exemple de traduction d'un `if else` emboîté. Dans cet exemple, les contraintes gardées emboîtées ont été réifiées c'est-à-dire que la contrainte $C_1 \rightarrow (C_2 \rightarrow C_3)$ est écrite $(C_1 \wedge C_2) \rightarrow C_3$.

int sum(int n){		
1 int s,i ;	**1**	$s_0 = 0$
		$i_0 = 0$
2 while (i<n) {	**2**	un passage dans la boucle
s = s+i ;		$i_0 < n_0 \rightarrow s_1 = s_0 + i_0 \wedge i_1 = i_0 + 1$
i = i+1 ;		$\neg(i_0 < n_0) \rightarrow s_1 = s_0 \wedge s_2 = s_0 \wedge s_3 = s_0$
}		$\neg(i_0 < n_0) \rightarrow i_1 = i_0 \wedge i_2 = i_0 \wedge i_3 = i_0$
		deux passages dans la boucle
		$(i_0 < n_0 \wedge i_1 < n_0) \rightarrow s_2 = s_1 + i_1 \wedge i_2 = i_1 + 1$
		$(i_0 < n_0 \wedge \neg(i_1 < n_0)) \rightarrow s_2 = s_1 \wedge s_3 = s_1$
		$(i_0 < n_0 \wedge \neg(i_1 < n_0)) \rightarrow i_2 = i_1 \wedge i_3 = i_1$
		trois passages dans la boucle
		$(i_0 < n_0 \wedge i_1 < n_0 \wedge i_2 < n_0) \rightarrow s_3 = s_2 + i_2 \wedge i_2 = i_2 + 1$
		$(i_0 < n_0 \wedge i_1 < n_0 \wedge \neg(i_2 < n_0)) \rightarrow s_3 = s_2$
		$(i_0 < n_0 \wedge i_1 < n_0 \wedge \neg(i_2 < n_0)) \rightarrow i_3 = i_2$
3 return s ;}	**3**	return s_3 ;

FIG. III.5 – Exemple d'instruction de boucle

Instruction de boucle *while* (c) b Nous transformons tout d'abord chaque boucle en la boucle *while* correspondante. Nous déplions ensuite la boucle en fonction d'un nombre maximum de dépliages fourni par l'utilisateur ou donné par la complexité dans le pire des cas de l'algorithme. Pour décrire tous les passages possibles dans la boucle, les contraintes du corps de la boucle sont gardées par les conditions d'entrée dans la boucle. Ce principe est proche de celui utilisé dans [13], mais les contraintes gardées remplacent les connecteurs logiques.

Soit max le nombre maximum des dépliages, soit σ_0 la fonction de renommage avant exécution du bloc b, soit σ_i la fonction de renommage après exécution de i itérations de b, soient $const_{b^i}$ les contraintes générées pour l'itération i du bloc b, et soient var_{def} les variables qui sont définies dans b. L'instruction de boucle *while* (c) b est traduite par l'ensemble qui contient pour chaque i de 1 à max les contraintes gardées suivantes :

1. $(\bigwedge_{k=0}^{i} ssa_{\sigma_k}(c)) \rightarrow const_{b^i}$
2. pour chaque variable $x \in var_{def}$
   $(\bigwedge_{k=0}^{i-1} ssa_{\sigma_k}(c) \wedge ssa_{\sigma_i}(\neg c)) \rightarrow \bigwedge_{k=\sigma_i(x)}^{\sigma_{max}(x)} x_k = \sigma_{k-1}(x)$

et la fonction de renommage σ devient σ_{max}.

La première contrainte correspond au chemin où l'on est entré i fois dans la boucle, quand c est vraie pour les renommages de 0 à i. La deuxième contrainte correspond au chemin où l'on est rentré $i-1$ fois dans la boucle, quand c est vraie de 0 à $i-1$ et fausse à l'itération i. Cette contrainte joue le rôle de ϕ-fonction en assurant le même nombre de renommages quel que soit le nombre de passages dans la boucle. Un exemple de boucle est donné dans la figure III.5 pour 3 dépliages (i.e. $max = 3$).

```
/*@ requires
      @ (\forall int i ; i>=0 && i<t.length-1 ; t[i]<=t[i+1])
  @ ensures
    @ (\result !=-1==>t[\result]==v) &&
    @ (\result==-1==> (\forall int k ; 0<=k && k<t.length ; t[k] != v))
@*/
```

Spécification JML de la recherche binaire

$t_0[0] <= t_0[1] \wedge t_0[1] <= t_0[2] \wedge t_0[2] <= t_0[3]$
$result \neq -1 \rightarrow t_0[result] = v_0$
$result = -1 \rightarrow t_0[0] \neq v_0 \wedge t_0[1] \neq v_0 \wedge t_0[2] \neq v_0 \wedge t_0[3] \neq v_0$

Traduction en contraintes pour t.length = 4

FIG. III.6 – Exemple de spécification JML

Expressions JML JML [33] est un langage du premier ordre qui permet de spécifier des pré-conditions, post-conditions, des assertions, variants ou invariants dans le code ainsi que des comportements normaux et des comportements en présence d'exception. Nous considérons les expressions JML élémentaires sur les types entiers et booléens, ainsi que les quantificateurs bornés `\forall` et `\exists`. Les expressions élémentaires sont traduites directement en contraintes. Les quantificateurs bornés sont traduits en une conjonction finie dans le cas du `\forall` et une disjonction finie dans le cas du `\exists`. En ce qui concerne les renommages SSA, les expressions analysées dans la spécification le sont en considérant leur renommage initial, tandis que celles analysées dans la partie `\requires` (post-condition) le sont en considérant le renommage initial pour les variables d'entrée (i.e. arguments) et leur renommage final pour la variable `\result` qui représente la valeur renvoyée par une fonction. Plus précisément, nous remplaçons la variable `\result` de la post-condition par le dernier renommage de l'expression retournée par l'instruction *return* du programme.

La figure III.6 donne la spécification de la recherche binaire d'un élément v dans un tableau t en JML et sa traduction en contraintes. La pré-condition exprime que le tableau est trié. La post-condition exprime que si le résultat est différent de -1 alors le résultat est l'indice de l'élément cherché, sinon, l'élément n'est pas dans le tableau.

III.3.2 Introduction de variables booléennes et résolution d'un système hybride

Un système de contraintes construit selon les règles présentées en III.3.1 ne peut pas toujours être résolu de façon efficace, en particulier quand les conditions des contraintes gardées portent sur des variables d'entrée qui sont quelconques.

Soit par exemple le programme simple de la figure III.7 qui calcule la valeur absolue de la soustraction de ses entrées i et j. Les deux premières contraintes du CSP associé sont issues de l'instruction conditionnelle, et la troisième de la négation de

//@ ensures $\backslash result \geq 0$	Domaines
public int simple(int i, int j) {	$D_{i0} = D_{j0} = D_{r0} = [-2^{f-1}, 2^{f-1}+1]$
if (i<j) return j-i ;	Contraintes
else return i-j ;	**1** $i_0 < j_0 \rightarrow r_0 = j_0 - i_0$
}	**2** $\neg(i_0 < j_0) \rightarrow r_0 = i_0 - j_0$
	3 $r_0 < 0$

FIG. III.7 – Exemple d'un programme simple (à gauche) et de sa traduction en CSP (à droite)

la post-condition, f est le format des entiers. Un solveur de contraintes en domaines finis peut filtrer le domaine de r_0 à $[-2^{f-1}, -1]$ par la contrainte 3. Mais puisque rien n'est connu sur les entrées i_0 et j_0 les contraintes gardées 1 et 2 ne permettent pas de réduire le domaine de i_0 et j_0. Ainsi, un processus d'énumération très coûteux est mis en jeu et l'inconsistance est détectée seulement quand le domaine de i_0 et j_0 a été réduit à une seule valeur.

Afin d'améliorer la résolution de tels systèmes de contraintes gardées, j'ai proposé une *approche hybride* qui combine booléens pour la partie contrôle et entiers pour la partie opérative. Les points clefs de cette approche hybride sont les suivants :

- Les gardes sont abstraites par des variables booléennes ; ainsi, les contraintes gardées peuvent être propagées dès que la variable booléenne a été affectée.
- Contrairement aux approches par model checking avec abstraction de prédicats (voir introduction section I.2.2 page 8), les variables du corps des contraintes gardées ne sont pas abstraites. Une instruction du type : «if (i< j) x = 4 ;» est donc traduite par une contrainte gardée «b $\rightarrow x = 4$» où « b » représente l'expression logique « i<j ».
- Le système hybride qui contient à la fois les booléens introduits pour les gardes et les variables entières de la partie opérative est résolu. L'énumération se fait d'abord sur les variables booléennes ce qui permet de propager les contraintes gardées et de réduire le domaine des variables entières.
- Quand le domaine des variables entières a été réduit, le mécanisme de *propagation inverse* des contraintes gardées (voir section III.2.1 page 96) élimine certaines solutions booléennes. Ceci réduit le nombre de solutions fallacieuses.

Cette approche permet donc de trouver rapidement des chemins admissibles dans le programme (grâce au solveur booléen) tout en gardant suffisamment d'informations sur les données entières (grâce au solveur entier) pour ne pas proposer trop de solutions fallacieuses.

L'exemple III.4 illustre comment la propagation inverse des contraintes gardées réduit le nombre de solutions fallacieuses.

Exemple III.4 (Résolution d'un système hybride) Soit le système hybride : $\{b_0 \rightarrow x_0 = 10, b_1 \rightarrow x_0 = 1\}$. La résolution de ce système hybride donne les deux solutions : $(b_0 = 0, b_1 = 1, x_0 = 1)$ et $(b_0 = 1, b_1 = 0, x_0 = 10)$. Ici le fait que le solveur utilise un mécanisme de propagation inverse des contraintes gardées est crucial . En

effet, affecter la valeur $b_0 = 1$ fixe la valeur $x_0 = 10$ par la première contrainte et le mécanisme de propagation inverse de la deuxième contrainte fixe la valeur $b_1 = 0$. Notons que si l'on avait également pris une abstraction de la partie opérative pour construire un système purement booléen, c'est-à-dire le système $\{b_0 \rightarrow b_2, b_1 \rightarrow b_3\}$, on aurait obtenu 9 solutions en tout dont 7 fallacieuses.♯

Notation Nous noterons $abstraction(exp)$ la variable booléenne qui abstrait l'expression exp.

La figure III.8 détaille l'algorithme de modélisation et de résolution hybride. Les points **1** et **2** correspondent à la construction du système de contraintes associé au programme et à sa spécification comme expliqué en III.3.1. Ce système est ensuite abstrait (point **3**) pour obtenir un système hybride dont la partie opérative est entière et la partie contrôle est booléenne. Ce système est résolu (point **4**) et pour chacune de ses solutions, s'il y en a, le système entier correspondant est construit (point **4.a**). Ce système entier contient deux parties : les contraintes issues des gardes (modélisées par des booléens dans le système hybride) et les contraintes entières issues de la partie opérative (qui apparaissaient en partie droite des contraintes gardées du système hybride). Notons que ce système entier ne contient pas de contraintes gardées puisque la solution booléenne a résolu les gardes ; il correspond donc à un chemin dans le programme. Notons aussi que le domaine initial des variables entières est celui qui a été réduit par résolution du système hybride. Si ce système entier a une solution (point **4.b**), alors cette solution est un cas d'erreur du programme puisqu'elle satisfait à la fois la pré-condition, un chemin dans le programme et la négation de la post-condition. Si le système entier n'a pas de solution, cela peut signifier deux choses. Soit la solution booléenne trouvée est fallacieuse, c'est-à-dire qu'en redonnant leur sémantique dans les entiers aux expressions qui ont été abstraites alors la solution booléenne trouvée est incohérente dans les entiers. Soit la solution booléenne est cohérente dans les entiers et c'est la partie opérative qui est inconsistante ce qui signifie que sur ce chemin, il n'existe pas de combinaisons d'entrée qui satisfont à la fois la pré-condition, les contraintes du chemin et la négation de la post-condition et le programme est correct sur ce chemin (ceci est illustré dans l'exemple III.3.3). Le point **5** correspond au cas où le programme est correct. En effet, soit le système hybride n'a pas de solutions, et donc a fortiori le système initial n'en a pas. Soit pour toutes les solutions booléennes du système hybride, le système entier correspondant est inconsistant ce qui signifie que la solution booléenne est fallacieuse. Si cela est vrai pour toutes les solutions booléennes, alors le programme est correct pour tous les chemins.

III.3.3 Exemple : vérification du programme *AbsMinus*

Le programme *AbsMinus* de la figure IV.3 calcule la valeur absolue de ses deux paramètres d'entrée i et j. La variable locale k n'a d'autre rôle que de complexifier légèrement le flot de contrôle de ce programme.

Le système hybride associé au programme *AbsMinus* est présenté dans la figure III.10. La résolution de ce système hybride donne une première solution ($b_0 = 0, b_1 =$

1. Mettre le programme en pseudo-forme SSA et le traduire en un ensemble de contraintes comme expliqué en III.3.1.
2. Ajouter les contraintes de la pré-condition et de la `négation` de la post-condition.
3. Introduire une variable booléenne pour chaque garde qui contient une variable d'entrée. On note $abstraction(g)$ la variable booléenne associée à la garde g

Soit *HybridSystem* le CSP obtenu après les étapes 1, 2, et 3.

4. Commencer la résolution de *HybridSystem* en énumérant sur les variables booléennes et pour *chaque* solution s :
  4.a. Construire le CSP *IntSystem* correspondant à s.
  Pour chaque variable entière de *IntSystem*, le domaine initial est celui obtenu dans s.
  Pour chaque contrainte gardée $abstraction(g) \rightarrow c$ de *HybridSystem*, si $abstraction(g) = 1$ dans s alors g et c sont ajoutés à *IntSystem*, sinon $\neg g$ est ajouté à *IntSystem*.
  4.b. Commencer la résolution de *IntSystem* et pour *chaque* solution de *IntSystem* afficher "le programme contient une erreur".
  Afficher la valeur des variables booléennes (trace du chemin) et afficher les solutions de *IntSystem* (valeur numérique).
5. Si *HybridSystem* n'a pas de solution ou si pour chaque solution booléenne *IntSystem* n'a pas de solution, afficher "le programme est conforme à sa spécification".

FIG. III.8 – Processus de vérification avec modélisation hybride

$0, b_2 = 0, r_0 = 0, k_0 = 0, k_1 = 0, r_1 = [-2^{f-1}, 2^{f-1} + 1], result_0 = [-2^{f-1}, 2^{f-1} + 1])$. Le système entier associé à cette solution est construit en introduisant la contrainte $abstraction(exp)$ si $b_i = 1$ et $\neg\ abstraction(exp)$ sinon. On obtient donc le système $\{\neg(i_0 \leq j_0), \neg(k_1 = 1 \wedge \neg(i_0 = j_0)), \neg(i_0 < j_0), r_0 = 0, k_0 = 0, k_1 = 0, r_1 = i_0 - j_0, result_0 = r_1, \neg result_0 = i_0 - j_0\}$ qui est trivialement inconsistant. Notons qu'ici le sous-système qui contient uniquement la sémantique des variables booléennes (i.e. $\{\neg i_0 \leq j_0, \neg(k_1 = 1 \wedge \neg(i_0 = j_0), \neg(i_0 < j_0)\}$ est consistant ce qui signifie que la solution booléenne n'est pas fallacieuse. Par contre, les contraintes du chemin plus celles de la négation de la post-condition sont inconsistantes ce qui signifie que le programme est correct sur ce chemin. Les autres solutions sont traitées de manière similaire. Nous pouvons remarquer que toute solution du système hybride pour laquelle $b_0 = 0$ et $b_1 = 1$ est une solution fallacieuse puisque cela correspond dans le système entier aux deux contraintes contradictoires $\neg i_0 \leq j_0$ et $i_0 < j_0$. Il aurait été plus judicieux ici d'introduire les booléens b'_0 pour abstraire $i_0 < j_0$ et b'_1 pour abstraire $i_0 = j_0$ comme discuté dans la section III.3.5.

```
1   class AbsMinus {
2   /*@ ensures   ((i < j) ==> (\result == j-i))
3               && ((i >= j) ==> (\result == i-j));  @*/
4   int absMinus (int i, int j) {
5     int r = 0;
6     int k = 0;
7     if (i <= j) k = k+1;
8     if (k == 1 && i != j) r = j-i;
9        else r = i-j;
10    return r;
11  }
12 }
```

FIG. III.9 – AbsMinus : programme de calcul de la valeur absolue d'une différence

1. Variables
   Variables entières : $\{i_0, j_0, r_0, r_1, k_0, k_1, result_0\}$
   Variables booléennes : $\{b_0, b_1, b_2\}$
2. Contraintes
   Contraintes issues du programme
   1 $r_0 = 0$
   2 $k_0 = 0$
   3 $b_0 \rightarrow k_1 = k_0 + 1$
   4 $!b_0 \rightarrow k_1 = k_0$
   5 $b_1 \rightarrow r_1 = j_0 - i_0$
   6 $!b_1 \rightarrow r_1 = i_0 - j_0$
   7 $result_0 = r_1$
   Contraintes issues de la négation de la post-condition
   8 $\neg((b_2 \rightarrow result_0 = j_0 - i_0) \wedge (!b_2 \rightarrow result_0 = i_0 - j_0))$
3. Table d'abstraction
   $abstraction(i_0 \leq j_0) = b_0$
   $abstraction(k_1 = 1 \wedge \neg i_0 = j_0) = b_1$
   $abstraction(i_0 < j_0) = b_2$

FIG. III.10 – Système hybride associé au programme *Absminus*

III.3.4 Résultats expérimentaux

Cette première approche a été implémentée en utilisant le solveur ILOG[3] solver (voir `http://www.ilog.com/products/solver`). Elle a été validée sur des exemples non triviaux, décrits dans [17] (voir aussi article support section III.6.1). En particulier, nous avons vérifié un programme de type contrôle/commande "tritype" qui contient de nombreux chemins non exécutables (voir figure III.14). Sur cet exemple, l'introduction de booléens a permis de résoudre le système efficacement en 2.36 secondes pour la vérification complète[4], alors que c'était impossible sur la version purement entière. Sur cet exemple, nous avons exploré seulement 92 solutions du système booléen alors qu'il y a 9 variables et donc potentiellement 2^9 solutions booléennes. Un autre point essentiel de cette approche, très utile dans le cas du programme "tritype" qui contient de nombreux chemins, est de fournir une trace très explicite en cas d'erreurs dans le programme. En effet, les valeurs des variables booléennes fixent les chemins et l'on peut donner pour chaque chemin plusieurs valeurs entières qui violent la post-condition.

Nous avons également vérifié des programmes de tri et de recherche binaire dans un tableau de façon plus efficace que d'autres approches basées sur du bounded model checking. En particulier, nous avons traité deux exemples de tri tirés de [5], un tri à bulles et un tri par insertion, où une pré-condition impose que les entrées soient triées en ordre décroissant. L'approche proposée dans [5] consiste à effectuer un bounded model checking en générant un système d'expressions conditionnelles qui est résolu par un solveur SMT. Notre approche a par exemple permis de vérifier le tri à bulles pour un tableau de taille 100 en moins d'une seconde, alors qu'il fallait $600s$ pour un tableau de taille 26 dans [5]. Ici, l'efficacité de notre méthode réside dans la collaboration des deux systèmes booléen et entier. La pré-condition fixe des valeurs constantes pour les éléments du tableau. Cela fixe aussi les conditions des gardes ce qui permet de propager les contraintes gardées et de calculer les valeurs entières des éléments du tableau. Ces valeurs sont utiles lors de la propagation inverse des contraintes gardées pour éliminer des solutions booléennes.

Nous ne nous attardons pas ici sur ces résultats qui ont été améliorés par notre deuxième approche par exécution symbolique le long du graphe de flot de contrôle du programme. Nous discutons dans la sous-section suivante de l'optimisation de la génération du système hybride et de l'apport d'un solveur SAT pour cette approche.

III.3.5 Discussion sur l'approche avec abstraction booléenne

Réalisations logicielles

Afin de mieux évaluer l'intérêt de l'approche par modélisation hybride, j'ai automatisé l'introduction des variables booléennes et appliqué cette approche à un plus grand ensemble d'exemples. Pour séparer la partie avant (i.e. traduction du langage de programmation) et le noyau de génération du CSP, j'ai défini la grammaire d'un sous-ensemble très restreint de Java (entiers, booléens et tableaux) sous la forme

[3] Ilog est désormais une société du groupe IBM.

[4] Ces expérimentations ont été effectuées sur un Intel(R) Pentium(R) 4 CPU 2.00GHz avec 256 Mb de mémoire.

d'une grammaire XML. J'ai ensuite implémenté un outil qui prend en entrée un programme et sa spécification décrits en XML, génère un système de contraintes hybride et le résout. Dans le même temps, j'ai encadré le projet de Master première année de Sébastien Derrien et Eric Le Duff, qui ont réalisé un plug-in eclipse pour traduire un programme Java et sa spécification JML en cette forme intermédiaire XML. Ce plug-in utilise l'API JDT d'Eclipse pour analyser les programmes Java [27] et un traducteur de JML fourni dans sa distribution standard [33].

Choix des expressions booléennes

Le programme *AbsMinus* a montré que le choix des expressions logiques qui sont représentées par des variables booléennes a une répercussion importante sur le nombre de solutions fallacieuses trouvées. Pour être efficace, il faut réduire le nombre de variables booléennes mais en introduire suffisamment pour ne pas perdre le lien sémantique entre deux sous-expressions. Les points clefs sont les suivants :

- Il est important de maintenir le lien entre des sous-expressions qui apparaissent dans différentes contraintes. Par exemple, l'expression $i \leq j$ doit être modélisée par $b_1 \vee b_2$ où b_1 représente $i < j$ et b_2 représente $i = j$. Ainsi, si l'expression $i < j$ apparaît dans une autre partie du programme, la variable b_1 sera aussi utilisée, et pourra permettre de résoudre une autre contrainte gardée.
- Il faut introduire une variable booléenne pour représenter l'expression renvoyée par le programme et la variable de la spécification qui représente le résultat. En effet, c'est le moyen d'établir un lien entre la spécification et le programme et cela permet de couper certains chemins.
- Quand les calculs sont complexes, il est aussi utile de représenter les expressions d'affectation par des variables booléennes.

Nous avons donc mis en œuvre des règles simples de réécriture des expressions à valeur booléenne afin d'utiliser seulement les opérateurs $<$ et $=$. Ainsi, l'expression $i > j$ est réécrite en $\neg((i < j) \vee (i = j))$. L'introduction des variables booléennes est faite à la volée en gérant une table de hachage.

L'exploration des différentes modélisations hybrides a été publiée et présentée à la conférence CP'2007 (Constraint Programming) à Providence en septembre 2007 [18]. La figure III.11 présente le système hybride le plus efficace pour le programme *AbsMinus* de la figure IV.3. L'expression $i \leq j$ a été remplacée par $i < j \vee i = j$ afin de générer moins de solutions fallacieuses. De plus, des booléens b_3 et b_4 ont été associés aux expressions renvoyées dans les différents cas d'utilisation du programme. b_3 est la valeur renvoyée quand $i < j$ et b_4 quand $\neg i < j$. La contrainte $b_3 + b_4 = 1$ exprime que ces deux variables booléennes ne peuvent être vraies en même temps. Les résultats expérimentaux reportés dans [18] montrent que cette modélisation est nettement plus efficace que la modélisation naïve de la figure III.10.

Utilisation du solveur booléen SAT4J

Une autre question que nous nous sommes posée concerne l'efficacité de la résolution de la partie booléenne du système hybride. En effet, nous utilisions un solveur CSP généraliste, pour lequel la résolution d'un système booléen est traité comme la résolution d'un système à domaines finis ayant les valeurs 0 et 1, avec

1. Variables
   Variables entières : $\{i_0, j_0, r_0, r_1, k_0, k_1, result_0\}$
   Variables booléennes : $\{b_0, b_1, b_2, b_3, b_4\}$
2. Contraintes
   Contraintes issues du programme
   1 $r_0 = 0$
   2 $k_0 = 0$
   3 $(b_0 \vee b_1) \rightarrow k_1 = k_0 + 1$
   4 $!(b_0 \vee b_1) \rightarrow k_1 = k_0$
   5 $b_2 \rightarrow b_0$
   6 $!b_2 \rightarrow b_1$
   7 $b_3 + b_4 = 1$
   Contraintes issues de la négation de la post-condition
   8 $\neg((b_0 \rightarrow b_3) \wedge (!b_0 \rightarrow b_4))$
3. Table d'abstraction
   $abstraction(i_0 < j_0) = b_0$
   $abstraction(i_0 = j_0) = b_1$
   $abstraction(k_1 = 1 \wedge \neg i_0 = j_0) = b_2 = k_1 = 1 \wedge b_1$
   $abstraction(b_0) = r_1 = j_0 - i_0$
   $abstraction(b_1) = r_1 = i_0 - j_0$

FIG. III.11 – Système hybride *optimisé* pour le programme *Absminus*

quelques optimisations spécifiques pour le solveur que nous avons utilisé. Nous avons donc voulu savoir si l'utilisation d'un solveur SAT augmenterait l'efficacité de résolution du système hybride. Nous avons effectué une première évaluation de l'utilisation du solveur SAT4J (SAT for Java) réalisé à l'Université de Lens, lors du projet de Master Recherche de Lydie Blanchet. SAT4J est une implémentation en Java de miniSAT, d'où notre choix pour faciliter son intégration dans notre suite d'outils écrits en Java. De plus, nous avions établi une collaboration avec Daniel Le Berre, un des concepteurs de SAT4J, qui a interagi avec Lydie tout au long de son projet. Le principe de collaboration implémenté est le suivant. Chaque expression entière (garde ou partie opérative) est associée à une variable booléenne. Le système purement booléen est d'abord résolu puis le système entier correspondant à la solution booléenne est construit puis résolu avec le solveur CSP.

Bien que ce solveur booléen soit très efficace, son utilisation n'a pas permis d'améliorer les performances de façon drastique. En effet, la traduction en forme normale conjonctive, format d'entrée du solveur SAT, ainsi que les appels à ce solveur ont un coût relativement élevé. De plus, les systèmes booléens générés pour les exemples que nous avons traités étaient trop petits pour vraiment bénéficier de l'intérêt d'un solveur SAT. Les premières conclusions ont montré qu'il faut à tout prix réduire le nombre de solutions booléennes correspondant à des chemins infaisables.

```
int foo(int x, int y) {
 1    int z = 0;                    1    z_0 = 0
 2    int r = 0;                    2    r_0 = 0
 3    if (x>y)                      3    b_0 → z_1 = 20
        z = 20;                          ¬b_0 → z_1 = 5
      else z = 5;
 4    if (z>10)                     4    b_1 → r_1 = x_0 + 1
        r = x+1;                         ¬b_1 → r_1 = 2 * y_0
      else r = 2*y;
 5    r++;                          5    r_2 = r_1 + 1
 return r;}
```

FIG. III.12 – Exemple de programme (à gauche) et de système hybride associé difficile à résoudre (à droite)

Pour cela, il faut propager les informations obtenues par le solveur entier dans le système booléen. D'autre part, un point critique pour l'efficacité reste le problème des convergences lentes lors de la résolution du système entier.

Défaut de la méthode

Si l'approche avec abstraction booléenne des conditions a permis de vérifier des exemples de taille significative, elle présente néanmoins une lacune majeure : l'ordre de parcours du graphe de flot de contrôle du programme est fortement dépendant de l'algorithme de résolution du système de contraintes. En effet, les chemins "résolus" grâce à l'énumération sur les variables booléennes ne le sont pas forcément dans l'ordre des instructions du programme. La procédure "foo" figure III.12 illustre ce problème.

Dans cet exemple, si $x > y$, alors le seul chemin exécutable pour l'instruction conditionnelle 4 est celui pour lequel $z > 10$. Cependant, lors de l'énumération sur les variables booléennes du système hybride, il est tout à fait possible d'obtenir l'instanciation $b_0 = 1$ et $b_1 = 0$. Cette solution sera fallacieuse et la résolution du système correspondant sur les entiers donnera une inconsistance. En effet, ce système contient les contraintes $\{z_0 = 0, r_0 = 0, x_0 > y_0, z_1 = 20, \neg z_1 > 10, r_1 = 2 * y_0\}$ où la contrainte $x_0 > y_0$ provient de $b_0 = 1$ et la contrainte $\neg z_1 > 10$ provient de $b_1 = 0$. Or les deux contraintes $z_1 = 20$ et $\neg z_1 > 10$ sont clairement inconsistantes. Nous allons voir que l'exécution symbolique en construisant le système de contraintes à la volée le long du graphe de flot de contrôle permet d'éviter ce problème.

III.4 Approche par exécution symbolique

En collaboration avec Pascal Van Hentenryck de l'Université de Brown et Michel Rueher, nous avons défini une nouvelle approche qui n'utilise pas de contraintes gardées. Le point fort de cette approche est de couper les chemins infaisables au

plus vite, tout en gardant le principe de la résolution d'un système hybride qui est une des originalités de nos travaux.

III.4.1 Exploration des chemins

Le principe général est le suivant. Nous parcourons successivement tous les chemins de *longueur bornée* dans le programme en générant un système de contraintes *à la volée*. Ce système contient les contraintes issues de la pré-condition et des décisions et instructions prises sur le chemin. Arrivé à la fin du chemin (i.e. quand la dernière instruction du programme est atteinte), nous ajoutons la négation de la post-condition et résolvons le système. Si ce système est inconsistant, alors le programme est correct pour ce chemin.

Ce principe est identique à celui présenté en III.1.2, sauf qu'il procède en construisant les CSP associés aux chemins d'exécution les uns après les autres, alors que l'approche précédente construisait un unique CSP qui décrivait tous les chemins d'exécution possibles.

> La question qui se pose est donc d'énumérer ces chemins de façon efficace. Dans notre approche précédente (voir section III.3 page 99), c'était la recherche de solutions du système booléen qui forçait l'ordre d'exploration des chemins : chaque solution booléenne correspondait à un chemin dans le programme, éventuellement non exécutable. Nous adoptons ici une démarche différente qui s'avère bien plus efficace : le système de contraintes associé à un chemin est construit *à la volée*. Ainsi, les chemins qui sont trivialement infaisables ne sont pas explorés. Pour cela, quand une instruction de contrôle est atteinte, nous testons si la condition est consistante avec le système de contraintes courant. En fonction du résultat de ce test, nous prenons l'un ou l'autre des chemins, en ajoutant les contraintes de la branche choisie : la condition ou sa négation et les contraintes issues des instructions de la branche choisie. Notons que le CSP généré pour chaque chemin ne contient pas de contraintes gardées ce qui a une conséquence directe en terme de rapidité de résolution du système de contraintes. De plus, les contraintes associées à un chemin sont construites en gérant un renommage SSA comme expliqué en III.3.1, mais ce renommage est beaucoup plus simple et efficace puisqu'il contient uniquement les cas d'affectation de variables scalaires ou de tableaux. Il n'est donc plus nécessaire d'introduire des contraintes pour jouer le rôle de ϕ-fonctions en assurant le même nombre de renommages dans les différentes branches des instructions de contrôle.

Ce principe de génération des CSP *à la volée* peut être considéré comme une *exécution symbolique* du programme. En effet, le système de contraintes représente de façon symbolique les affectations et décisions de la partie du chemin qui a été exécutée dans le programme. Nous emploierons donc le terme *d'exécution symbolique* par la suite ; les principes en sont détaillés dans la section suivante.

III.4.2 Algorithme d'exécution symbolique

L'algorithme de vérification par exécution symbolique est décrit figure III.13. Il prend en entrée la pré-condition *pre*, la post-condition *post*, la liste d'instructions à

exécuter l et la borne maximale de dépliage des boucles b qui permet de borner la longueur des chemins explorés. La fonction $verify(pre, post, l, b)$ renvoie un booléen qui indique si l est conforme à sa spécification c'est-à-dire que le triplet de Hoare $\{pre\}l_i\{post\}$ est vérifié pour tous les chemins l_i dans l dont le nombre de passage dans les boucles est borné par b.

Dans la figure III.13, la fonction $execSymb(post, l, csp, b)$ définit l'exécution symbolique du programme l. csp représente l'état courant de l'exécution symbolique puisqu'il contient les contraintes associées aux affectations et décisions rencontrées jusqu'à l'instruction courante. Ce CSP a été initialisé avec les contraintes associées à la pré-condition pre. L'instruction conditionnelle `if else` est notée $(Cond\ c\ i_{if}\ i_{else})$, où c est la condition et i_{if} et i_{else} sont respectivement les instructions de la branche if et $else$. L'instruction de boucle `while` est notée $(While\ c\ i_{while})$ où c est la condition et i_{while} les instructions du corps de la boucle. Pour chaque instruction de boucle i, $depliage(i)$ est le nombre de dépliages courant de i[5]. $contr(c)$ (resp. $contr(i)$) représente les contraintes associées à la condition c d'une instruction de contrôle (resp. d'une instruction i). Cette fonction gère aussi de façon implicite les renommages SSA, comme expliqué en III.3 page 98.

La fonction $execSymb$ fait appel à deux fonctions qui gèrent les appels au solveur. Étant donné csp qui contient les contraintes de la pré-condition et du chemin courant, $test_{cond}(c, csp)$ teste la consistance de la condition c sur ce chemin et $test_{chemin}(post, csp)$ teste la validité du programme sur ce chemin. Ces fonctions renvoient une valeur booléenne :

- $test_{cond}(c, csp)$ renvoie vrai si $c \wedge csp$ est satisfiable. Cela signifie qu'une combinaison des entrées qui permet d'atteindre une instruction conditionnelle satisfait aussi sa condition c. Il faut donc explorer le chemin correspondant.
- $test_{chemin}(post, csp)$ renvoie vrai si $\neg post \wedge csp$ est inconsistant. Cela signifie que le programme est correct sur le chemin considéré puisqu'il n'existe pas de valeurs qui satisfont la pré-condition, les contraintes du chemin et la négation de la post-condition.

Dans l'algorithme de la figure III.13, le point **1** est le cas de terminaison : la fin du programme est atteinte et il faut tester la correction de ce chemin. Le point **2** est le cas d'une instruction simple : on ajoute les contraintes correspondantes (i.e. $contr(i)$) et on continue l'exécution symbolique sur les instructions suivantes (i.e. liste d'instructions l'). Le point **3** est le cas d'une instruction de contrôle. L'état courant est d'abord sauvegardé car l'exécution symbolique peut explorer les deux branches. En effet, si le test de la condition porte sur des variables d'entrée qui ont des valeurs quelconques, il peut exister des valeurs qui satisfont la condition et d'autres valeurs qui satisfont la négation de la condition. Les deux chemins sont donc exécutables et il faut alors effectuer un retour arrière pour prendre le deuxième chemin. Le point **4** est le cas où la condition est possible. Il y a alors deux sous-cas, points **4a** et **4b**, selon qu'il s'agisse d'une instruction conditionnelle ou d'une instruction de boucle. Dans les deux cas, on ajoute la condition au CSP (i.e. csp devient $csp \wedge contr(c)$). Pour une conditionnelle, point **4a**, on exécute la branche if suivie du reste du programme (i.e. liste d'instructions $[i_{then}, l']$). Pour une boucle,

[5]Ceci est géré très facilement en associant un identificateur aux instructions de boucles.

point **4b**, on teste d'abord si l'on a atteint le nombre maximum de dépliages. Si c'est le cas, on s'arrête en affichant un message d'erreur, sinon on rentre dans la boucle c'est-à-dire que l'on exécute symboliquement la liste $[i_{while}, (\texttt{While}\ c\ i_{while}), l']$). Dans le cas où la condition est possible, il se peut aussi que sa négation le soit ; il faut alors considérer ce cas. On commence donc par restaurer le contexte d'avant traitement du chemin pris quand la condition est supposée vraie. Puis on teste si la négation de la condition est possible. Si c'est le cas, on procède comme décrit dans le point **5**.

`verify`$(pre, post, l)$ =
  `execSymb`$(post, l, contr(pre), b)$

`execSymb`$(post, l, csp, b)$ =
1. Si `l=[]` la fin d'un chemin est atteinte et le résultat est $test_{chemin}(post, csp)$
Sinon soit `l = [i,l']`
  2. Si `i` **n'est pas une instruction de contrôle**, appel récursif de `execSymb`$(post, l', csp \wedge contr(i), b)$ pour continuer l'exécution symbolique
  3. Sinon soit c **la condition** de i
    - invoquer $test_{cond}(c, csp)$ pour savoir si c est consistante sur le chemin courant
    4. Si c **est consistante**
      - sauvegarder l'état actuel des compteurs de dépliage et de renommage SSA
      - prendre le chemin correspondant dans le programme :
        4a. Si i **est la conditionnelle** `(Cond` $c\ i_{then}\ i_{else}$`)`
          appel récursif de `execSymb`$(post, [i_{then}, l'], csp \wedge contr(c), b)$
          pour exécuter la branche `then`
        4b. Si i **est l'instruction de boucle** `(While` $c\ i_{while}$`)`
          Si $depliage(i) < b$ appel récursif de
            `execSymb` $(post, b, [i_{while}, (\texttt{While}\ c\ i_{while}), l'], csp \wedge contr(c))$
            pour entrer dans la boucle ; mettre à jour $depliage(i)$.
          Sinon afficher que le nombre de dépliages maximum est insuffisant et renvoyer **faux**.
      - restaurer l'état des compteurs de dépliage et de renommage SSA
      - invoquer $test_{cond}(\neg c, csp)$ pour savoir si la **négation** de la condition est également possible
      - si $\neg c$ est consistante, prendre le chemin correspondant dans le programme comme expliqué en **5**
    5. Sinon (i.e. c est inconsistante)
      5a. Si c **est la conditionnelle** `(Cond` $c\ i_{then}\ i_{else}$`)` alors appel récursif de `execSymb`$(post, [i_{else}, l'], csp \wedge contr(\neg c), b)$ pour exécuter la partie `else`
      5b. Si c **est l'instruction de boucle** `(While` $c\ i_{while}$`)` appel récursif de `execSymb`$(post, l', csp \wedge contr(\neg c), b)$ pour sortir de la boucle.

FIG. III.13 – Algorithme d'exécution symbolique

Le point **5** est le cas où la condition est impossible. On ajoute donc la négation de la condition au CSP. Pour une conditionnelle, point **5a**, on exécute la branche *else*

suivie du reste du programme (i.e. liste d'instructions $[i_{else}, l']$). Pour une boucle, point **5b**, on sort de la boucle en exécutant le reste du programme (i.e. liste d'instructions l').

Remarque Pour accroître l'efficacité, nous postons en fait la négation de la post-condition dès le départ (i.e.. appel à `execSymb`$(post, b, l, contr(pre) \wedge contr(\neg post))$), et vérifions successivement tous les cas d'utilisation décrits dans la post-condition. Ceci nous permet de couper au plus vite les chemins qui ne correspondent pas au cas d'utilisation sélectionné. Par exemple, la post-condition du programme *AbsMinus* figure IV.3 correspond à deux cas d'utilisation pour $i < j$ et $i \geq j$. En prenant la négation de cette post-condition, on obtient une disjonction. Nous vérifions les deux cas de cette disjonction successivement. Pour le premier cas, les chemins qui ne satisfont pas $i < j$ seront coupés, et dans le deuxième cas, les chemins qui ne satisfont pas $i \geq j$ seront coupés. Notons que la gestion des renommages SSA dans la post-condition est effectuée en utilisant les renommages initiaux jusqu'à ce que la fin du programme soit atteinte ; à ce moment, on effectue le lien entre la variable `\result` et le dernier renommage de la valeur renvoyée par le programme. La vérification du programme *AbsMinus* est détaillée ci-dessous.

III.4.3 Exemple : vérification du programme *AbsMinus*

La vérification du programme *AbsMinus* de la figure IV.3 par exécution symbolique procède comme suit. Dans les CSP, toutes les variables ont pour domaine initial $[-2^{f-1}, 2^{f-1} - 1]$ où f est le format des entiers. Pour faciliter la lecture, nous considérons ici que $f = 8$ et donc les domaines des variables sont $[-127, 128]$.

Tout d'abord les contraintes associées à la négation de la post-condition sont calculées. On obtient la disjonction : $((i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0)) \vee ((i_0 \geq j_0) \wedge \neg(result_0 = i_0 - j_0))$. On vérifie successivement les deux cas de la disjonction.

Cas 1 : $(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0)$

1. La spécification est ajoutée au système de contraintes : $csp = \{(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0)\}$
2. Les instructions 1 et 2 sont traduites par les contraintes $r_0 = 0$ et $k_0 = 0$ pour obtenir $csp = \{(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0), r_0 = 0, k_0 = 0\}$
3. L'instruction conditionnelle 7 est exécutée en testant si la condition $i_0 \leq j_0$ est consistante par rapport à *csp*. Le système de contraintes $\{(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0), r_0 = 0, k_0 = 0, i_0 \leq j_0\}$ est résolu et donne trivialement la solution $r_0 = 0, k_0 = 0, i_0 = -127, j_0 = -126, result_0 = -127$. Le chemin correspondant est donc exploré et la contrainte $k_1 = k_0 + 1$ et la décision $i_0 \leq j_0$ sont ajoutées à *csp*.
4. L'instruction conditionnelle 8 est exécutée en testant si la condition $k_1 = 1 \wedge \neg(i_0 = j_0)$ est consistante par rapport à *csp*. Le système de contraintes $\{(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, k_1 = k_0 + 1, k_1 = 1 \wedge \neg(i_0 = j_0)\}$ est résolu et donne trivialement la solution $r_0 = 0, k_0 = 0, k_1 = 1, i_0 = -127, j_0 = -126, result_0 = -127$. Le chemin correspondant est donc

exploré et la contrainte $r_1 = j_0 - i_0$ et la décision $k_1 = 1 \wedge \neg(i_0 = j_0)$ sont ajoutées à *csp*.

5. La dernière instruction du programme (ligne 10) est atteinte. Il faut alors effectuer le lien entre la variable `\result` de la spécification et la valeur retournée par le programme. Pour des raisons d'efficacité, plutôt que d'introduire une contrainte qui établit l'égalité entre le dernier renommage de r et $result_0$, nous effectuons une substitution. Le système obtenu après exploration du premier chemin est le suivant : $\{(i_0 < j_0) \wedge \neg(r_1 = j_0 - i_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, k_1 = k_0 + 1, k_1 = 1 \wedge \neg(i_0 = j_0), r_1 = j_0 - i_0\}$. Ce système est trivialement inconsistant et le chemin correspondant dans le programme est donc correct.
6. Il faut maintenant explorer la deuxième branche de l'instruction conditionnelle ligne 8, en testant si la négation de la condition est consistante. $\{(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, k_1 = k_0 + 1, \neg(k_1 = 1) \vee i_0 = j_0\}$ est résolu. Ce système est trivialement inconsistant : puisque $k_1 = 1$ il faut que $i_0 = j_0$ ce qui est impossible par rapport au cas de la post-condition en cours de traitement (i.e. $(i_0 < j_0) \wedge \neg(result_0 = j_0 - i_0)$).
7. L'exécution symbolique revient donc au point de choix précédent de l'instruction conditionnelle 7, en testant si $\neg(i_0 \leq j_0)$ est consistante. Ceci est clairement inconsistant par rapport au cas de la post-condition en cours de traitement. Ceci termine donc ce premier cas. Un chemin a été exploré et il est correct.

Il nous faut maintenant vérifier le programme pour son deuxième cas d'utilisation.

Cas 2 : $\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0)$

1. La spécification est ajoutée dans le système de contraintes : $csp = \{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0)\}$
2. Les instructions 1 et 2 sont traduites en contraintes et ajoutées $csp = \{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0\}$
3. L'instruction conditionnelle 7 est exécutée en testant si la condition $i_0 \leq j_0$ est consistante par rapport à *csp*. Le système de contraintes $\{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0, i_0 \leq j_0\}$ est résolu et donne trivialement la solution $r_0 = 0, k_0 = 0, i_0 = -127, j_0 = -127, result_0 = -127$. Le chemin correspondant est donc exploré et la contrainte $k_1 = k_0 + 1$ et la décision $i_0 \leq j_0$ sont ajoutés à *csp*.
4. L'instruction conditionnelle 8 est exécutée en testant si la condition $k_1 = 1 \wedge \neg(i_0 = j_0)$ est consistante par rapport à *csp*. Le système de contraintes $\{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, k_1 = k_0 + 1, k_1 = 1 \wedge \neg(i_0 = j_0)\}$ est résolu. Il n'a pas de solution puisqu'ici encore ce chemin est incompatible avec le cas de la post-condition en cours de vérification. La deuxième branche est donc explorée en ajoutant la contrainte $r_1 = i_0 - j_0$ et la décision $\neg(k_1 = 1) \vee i_0 = j_0$.
5. La dernière instruction du programme est atteinte avec comme système final : $\{\neg(i_0 < j_0) \wedge \neg(r_1 = i_0 - j_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, k_1 = k_0 + 1, \neg(k_1 =$

$1) \vee i_0 = j_0, r_1 = i_0 - j_0\}$. Ce système est trivialement inconsistant et ce chemin est donc correct.

6. L'exécution symbolique explore maintenant la deuxième branche de l'instruction conditionnelle 7 en testant le système $\{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0, \neg(i_0 \leq j_0)\}$. Cette condition est possible et la décision $\neg(i_0 \leq j_0)\}$ est ajoutée à *csp*.
7. L'instruction conditionnelle 8 est exécutée en testant si la condition $k_0 = 1 \wedge \neg(i_0 = j_0)$ est consistante par rapport à *csp*[6]. Le système de contraintes $\{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0, \neg(i_0 \leq j_0), k_0 = 1 \wedge \neg(i_0 = j_0)\}$ est résolu. Il n'a pas de solution puisque $k_0 = 0$ et $k_0 = 1$ sont contradictoires. On explore donc l'autre branche qui est possible et donne le système $\{\neg(i_0 < j_0) \wedge \neg(result_0 = i_0 - j_0), r_0 = 0, k_0 = 0, \neg(i_0 \leq j_0), \neg(k_0 = 1) \vee i_0 = j_0, r_1 = i_0 - j_0\}$ après ajout de la décision et de la contrainte de la branche *else*.
8. La dernière instruction du programme est atteinte avec comme système final : $\{\neg(i_0 < j_0) \wedge \neg(r_1 = i_0 - j_0), r_0 = 0, k_0 = 0, i_0 \leq j_0, \neg(k_0 = 1) \vee i_0 = j_0, r_1 = i_0 - j_0\}$. Ce système est trivialement inconsistant et ce chemin est donc correct. Ceci termine la vérification de ce deuxième cas.

III.4.4 Résolution hybride

L'exemple du programme *AbsMinus* a mis en évidence l'intérêt d'une exécution symbolique puisque 3 chemins seulement parmi les 8 possibles (i.e. 4 chemins dans le programme et 2 cas d'utilisation) ont été explorés entièrement. Cependant, cette approche n'a de sens que si le test de consistance des conditions des instructions de contrôle est très efficace, puisqu'il est répété très fréquemment. Pour cela, nous avons repris le point clef de notre précédente approche : *la résolution hybride*.

> Nous utilisons de façon incrémentale plusieurs solveurs, en partant du plus spécifique et plus rapide vers le plus général et donc plus lent. L'implémentation actuelle intègre quatre solveurs : un module de simplification formelle chargé de réduire les expressions constantes et d'effectuer des réécritures simples[7], un solveur linéaire, un solveur booléen et un solveur en domaines finis qui procède par filtrage et énumération comme expliqué en III.2.1.

Plus précisément, nous gérons trois systèmes de contraintes :
- $csp_{complet}$ contient les contraintes issues de la pré-condition et des instructions et décisions du chemin,
- $csp_{lineaire}$ est un sous-ensemble de $csp_{complet}$ qui contient les contraintes linéaires,
- csp_{bool} contient une abstraction booléenne des contraintes non linéaires des décisions prises sur le chemin. Nous notons $abstraction(c)$ la ou les contraintes booléennes correspondant à c. Il s'agit soit d'une nouvelle variable booléenne,

[6]Notons qu'ici le renommage courant de k est k_0 puisque la branche if n'a pas été exécutée et donc k a été défini seulement une fois lors de sa création ligne 6.

quand l'expression et ses sous-expressions n'ont pas déjà été rencontrées, soit d'une combinaison logique de variables booléennes existantes[8].

L'idée est la suivante : les solveurs booléens et linéaires sont beaucoup plus efficaces que le solveur à domaines finis, puisqu'ils ne procèdent jamais par énumération. De plus, $csp_{lineaire}$ et csp_{bool} sont des relaxations de $csp_{complet}$. Ainsi, si $csp_{lineaire}$ ou csp_{bool} sont inconsistants, cela implique que $csp_{complet}$ l'est aussi. Dans le cas du test d'une condition c, si $csp_{lineaire} \wedge c$ ou $csp_{bool} \wedge c$ sont inconsistants, il ne faut pas prendre le chemin correspondant à c. Par contre, si $csp_{lineaire} \wedge c$ et $csp_{bool} \wedge c$ sont consistants, cela n'implique rien sur $csp_{complet} \wedge c$. Plutôt que de vérifier la consistance sur $csp_{complet}$ ce qui pourrait entraîner au pire un processus d'énumération coûteux, nous choisissons d'explorer les deux chemins (i.e. cas où la condition c est ou n'est pas vérifiée). Pour tester la correction à la fin d'un chemin, si $csp_{lineaire}$ ou csp_{bool} sont inconsistants alors on sait que $csp_{complet}$ l'est aussi, et le programme est correct sur ce chemin. Si $csp_{lineaire}$ et csp_{bool} sont consistants, alors on teste la consistance de $csp_{complet}$ pour décider si le programme est correct pour ce chemin.

Notre heuristique explore donc certains chemins inutilement plutôt que d'invoquer le solveur à domaines finis à chaque fois qu'une instruction de contrôle est atteinte. C'est seulement à la fin des chemins que ce solveur est appelé puisqu'il est indispensable dans ce cas de montrer l'inconsistance de $csp_{complet}$ si elle n'a pu être détectée sur $csp_{lineaire}$ et csp_{bool}.

Dans l'algorithme de la figure III.13, csp est en fait un triplet $csp = (csp_{lineaire}, csp_{bool}, csp_{complet})$. Toutes les contraintes sont ajoutées dans $csp_{complet}$. Si une contrainte est linéaire, elle est ajoutée aussi dans $csp_{lineaire}$, sinon son abstraction est ajoutée dans csp_{bool}. Les fonctions $test_{cond}(c, csp)$ et $test_{chemin}(post, csp)$ de l'algorithme figure III.13 utilisent la simplification formelle et les trois solveurs.

$test_{cond}(c, csp)$ est défini de la façon suivante :

- si c se réduit par simplification formelle à vrai ou faux, alors la fonction renvoie la valeur correspondante,
- si c est linéaire et que $csp_{lineaire} \wedge c$ est inconsistant alors la fonction renvoie faux,
- si c n'est pas linéaire et que $csp_{bool} \wedge abstraction(c)$ est inconsistant alors la fonction renvoie faux,
- sinon la fonction renvoie vrai.

$test_{chemin}(post, csp)$ est défini de la façon suivante :

- si $csp_{lineaire}$ est inconsistant alors $test_{chemin}$ renvoie vrai,
- sinon, si csp_{bool} est inconsistant alors $test_{chemin}$ renvoie vrai,
- sinon, $test_{chemin}$ renvoie la négation du résultat de la résolution de $csp_{complet}$.

III.4.5 Résultats expérimentaux

Cette section résume les résultats expérimentaux qui sont donnés de façon détaillée dans l'article de la conférence CP'2008 [19] inclus comme article support section III.6.2 page 143. Une version étendue de cet article est en cours de révision pour la revue "Constraints, An International Journal".

[8] Nous utilisons le même principe de décomposition des expressions booléennes que présenté en III.3.2.

Outils évalués

Nous avons comparé notre outil (noté CPBPV pour "Constraint Programming for Bounded Program Verification") à cinq outils différents :

- **CBMC** : "Bounded Model Checker for ANSI-C and C++". CBMC est un outil de bounded model checking très utilisé ; il permet de vérifier des problèmes de sûreté (e.g. sortie des bornes d'un tableau, accès correct aux pointeurs, ...) et d'assertions définies par l'utilisateur. Voir `http://www.cprover.org/cbmc`.
- **EUREKA** : un outil qui effectue du bounded model checking en utilisant un solveur SMT à la place d'un solveur SAT. Voir `http://www.ai-lab.it/eureka`.
- **BLAST** : "Berkeley Lazy Abstraction Software Verification Tool", un outil qui effectue du model checking de programmes C. Voir `http://mtc.epfl.ch/software-tools/blast`.
- **ESC/Java** : "Extended Static Checker for Java" qui détecte des erreurs d'exécution dans des programmes Java annotés en JML en faisant de l'analyse statique du code et de ses annotations. Voir `http://kind.ucd.ie/products/opensource/ESCJava2`.
- **Why** : une plate-forme de vérification de programmes qui intègre plusieurs assistants de preuve comme Coq, PVS, HOL 4,... ainsi que des procédures de décision comme Simplify, Yices, ... Voir `http://why.lri.fr`.

Ces outils partagent les caractéristiques suivantes :

- Ils prennent en entrée un sur-ensemble du langage impératif que nous pouvons vérifier (voir figure III.1) en utilisant des pré-conditions et post-conditions, même si certains sont dédiés à la vérification de programmes plus spécifiques (e.g., CBMC peut traiter des programmes Verilog et des formules en logique temporelle).
- Ils effectuent une analyse statique automatique,
- Une distribution libre est disponible, que l'on a pu télé-charger pour évaluer leurs performances sur notre ensemble d'exemples.

Ces outils sont cependant basés sur des méthodes très différentes qui recouvrent une grande partie des techniques de vérification formelle des programmes (voir [23] pour un état de l'art récent et complet des méthodes automatiques) :

- CBMC [13, 15] et EUREKA [5, 6] effectuent du bounded model checking. Ce sont les outils les plus proches de CPBPV puisqu'ils explorent tous les chemins d'une longueur maximale donnée. La différence principale est qu'ils génèrent une formule conditionnelle, purement booléenne pour CBMC et combinant booléens et entiers pour EUREKA ; cette formule est résolue par un solveur SAT pour CBMC et un solveur SMT pour EUREKA. C'est donc l'algorithme mis en œuvre dans le solveur SAT ou SMT qui permet de couper les chemins impossibles, alors que dans notre approche, nous utilisons les informations du chemin courant pour savoir si la condition est possible.
- BLAST [7] est basé sur du model checking et il implémente donc un processus d'abstraction / raffinement. La caractéristique principale de BLAST est d'utiliser une *abstraction paresseuse* : l'étape de raffinement est activée seulement

sur la partie concernée du programme.
- ESC/JAVA [16, 12] fait de l'analyse statique et nécessite que des annotations comme des invariants de boucle soient fournies par l'utilisateur.
- Why [25] est assez différent des outils précédents puisqu'il génère des obligations de preuve en calculant la plus faible pré-condition du programme. Le mécanisme de génération des obligations de preuve est basé sur la sémantique du langage orienté preuve "Why" qui a été définie et prouvée avec l'assistant de preuves Coq. L'outil génère les obligations de preuve dans les formats d'entrée de différents assistants de preuve (Coq, HOL4, Isabelle/HOL) et différentes procédures de décisions automatiques (Yices, siplify, CVC-lite, ...).

Bien que reposant sur des techniques très différentes, il nous a paru intéressant de comparer CPBPV à ces outils en terme d'efficacité, d'intervention de l'utilisateur, et de retour d'information quand une erreur est détectée.

Les résultats expérimentaux ont été effectués sur la même machine, un processeur Intel(R) Pentium(R) M 1.86GHz avec 1.5G de mémoire. La version des outils de vérification est celle qui était télé-chargeable en juin 2008 à partir de leur site Web (excepté pour EUREKA pour lequel les temps d'exécution sont ceux donnés dans [4, 5]). Pour CPBPV, l'implémentation utilisée intègre le solveur MIP (Mixed Integer Programming) d'ILOG comme solveur entier et le solveur JSOLVER d'ILOG comme solveur de contraintes généraliste[9].

Résultats

Recherche dichotomique dans un tableau Il s'agit de chercher l'indice d'un élément dans un tableau ordonné. La pré-condition est que le tableau est trié par ordre croissant, et la post-condition est que la valeur retournée est -1 si la valeur cherchée n'est pas dans le tableau et i si la valeur cherchée se trouve à l'indice i.

CPBPV a permis de vérifier ce programme pour un tableau de longueur 32 en $4.043s$ et nous avons pu prouver une instance de longueur 256. CBMC n'a pas pu effectuer la vérification pour un tableau de longueur 32 (interruption après $6691,87s$). Comme attendu puisqu'il repose sur un calcul de plus faible pré-condition, le système Why ne peut pas vérifier ce programme si on ne lui fournit pas un invariant de boucle : 60% des obligations restent inconnues. Par contre, le programme avec les assertions de boucle qui est fourni dans la distribution de Why est vérifié en 11.18s, et ce quelle que soit la longueur du tableau, puisqu'il effectue une vérification complète. De même, ESC/Java nécessite des invariants complets (sur les valeurs du tableau et des indices). Enfin, la version de BLAST utilisée ne peut pas effectuer cette vérification à cause de l'expression non linéaire du calcul du milieu.

> Nous avons également vérifié une version erronée de ce programme où une erreur de copier/coller fait que les indices *gauche* et *droit* sont modifiés de la même façon. Le point à souligner est que CPBPV fournit une trace d'erreur qui donne des valeurs numériques aussi bien pour le tableau que pour la valeur cherchée. Par contre, les traces fournies par CBMC et ESC/Java montrent seulement les décisions prises le long du chemin erroné (i.e. les valeurs des indices *gauche* et *droite*).

[9] Voir `http://www.ilog.com/products`.

Programme Tritype de classification d'un triangle Le programme Tritype est un benchmark classique pour la génération de jeux de test car il contient de nombreux chemins infaisables : seulement 10 chemins (sur 57) correspondent à des entrées effectives à cause d'une structure de contrôle complexe. Ce programme prend trois entiers positifs en entrée (les côtés du triangle) et renvoie 2 si le triangle est isocèle, 3 s'il est équilatéral, 1 si cela correspond à un autre triangle, et 4 sinon. La figure III.14 présente le programme tritype et sa spécification. La variable locale "trityp" détermine le nombre de côtés égaux et quels sont ces côtés.

CPBPV est très efficace sur cet exemple : 0.287s par raport à 0.82s pour CBMC ou 8.85s pour Why.

> En particulier, les temps d'exécution sont bien meilleurs que ceux obtenus pour l'approche par abstraction booléenne (8.52 secondes, voir III.3.4). Cela s'explique aisément par le fait que dans le programme Tritype les conditionnelles portent sur les variables d'entrée et la variable locale *trityp*. La résolution du système hybride donne donc des contre-exemples fallacieux.
> Par exemple, les instructions 7, 8 et 9 dans le programme sont traduites par les contraintes gardées avec abstraction booléenne suivantes : $b_0 \rightarrow trityp_2 = trityp_1 + 1, b_1 \rightarrow trityp_3 = trityp_2 + 1, b_2 \rightarrow trityp_3 = trityp_2 + 1$ où $abstraction(i = j) = b_0, abstraction(i = k) = b_1, abstraction(j = k) = b_2$. Avec une telle modélisation abstraite, le solveur booléen peut trouver la solution $b_0 = 1, b_1 = 1, b_2 = 0$ alors que ceci est sémantiquement impossible si l'on considère les expressions entières associées. Avec l'approche par exécution symbolique, une fois que les décisions $i = j$ et $i = k$ ont été prises, lorsque la condition $j = k$ est atteinte, le test de sa négation est inconsistant et nous explorons donc un seul chemin.

Nous avons également considéré une version incorrecte du programme Tritype dans laquelle le test $if((trityp == 2)\&\&(i+k > j))$ de la ligne 22 (voir figure III.14) a été remplacé par $if((trityp == 1)\&\&(i+k > j))$. Puisque la variable locale *trityp* est égale à *2* quand *i=k*, la condition $(i + k) > j$ implique que *(i,j,k)* sont les côtés d'un triangle isocèle. En effet, les deux autres inégalités triangulaires sont triviales puisque $i = k$ et $j > 0$. Mais quand *trityp=1* comme dans la version erronée, alors *i==j* et cette version incorrecte peut trouver que le triangle est isocèle alors qu'il peut ne pas être un triangle du tout (vérifier $(i+k) > j$ n'est pas suffisant dans ce cas, il faut aussi tester les autres inégalités triangulaires). Par exemple, la combinaison d'entrées *(i,j,k)=(1,1,2)* est un cas où le programme renvoie *2* (triangle isocèle) alors que ces valeurs ne correspondent pas aux côtés d'un triangle ($1 + 1 > 2$ est faux).

L'erreur trouvée par CPBPV correspond aux valeurs $(i, j, k) = (1, 1, 2)$ mentionnées précédemment et correspond donc au cas où le triangle est isocèle.

> Il est par ailleurs intéressant de constater que si l'on cherche plusieurs erreurs, 3 par exemple, CPBPV fournira les valeurs $(i, j, k) = (1, 1, 2)$, $(i, j, k) = (2, 2, 1)$ et $(i, j, k) = (3, 3, 1)$ ce qui peut aider l'utilisateur à comprendre que l'erreur se trouve dans un chemin où $i = j$. Les autres outils (e.g., ESC/JAVA et BLAST) fournissent des informations sur le chemin mais ne fournissent pas les valeurs numériques correspondantes.

```
/*@ requires (i>=0)&&(j>=0)&&(k>=0);
 @ ensures
 @   ((i+j<=k)||(j+k<=i)||(i+k<=j)) ==> \\ result == 4 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&((i==j)&&(j==k)) ==> \\ result == 3 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&((i==j)||(j==k)||(i==k)) ==> \\ result == 2 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&!((i==j)||(j==k)||(i==k)) ==> \\ result == 1;
@*/
1  public static int tritype(int i, int j, int k){
2   int trityp ;
3   // not a triangle
4   if ((i==0)||(j==0)||(k==0)) trityp = 4 ;
5   else {
6     trityp = 0 ;
7     if (i==j) trityp = trityp + 1 ;
8     if (i==k) trityp = trityp + 2 ;
9     if (j==k) trityp = trityp + 3 ;
10    if (trityp==0){
11       // triangular inequality not verified
12       if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4 ;
13       else trityp = 1 ; // any triangle
14    }
15    else {
16      if (trityp > 3) trityp = 3 ; // equilateral
17      else
18        //i=j and triangular inequality verified
19        if ((trityp==1)&&(i+j>k)) trityp = 2 ;
20        else
21          //i=k and triangular inequality verified
22          if ((trityp==2)&&(i+k>j)) trityp = 2 ;
                                //ERROR if ((trityp==1)&&(i+k>j))
23          else
24            //j=k and triangular inequality verified
25            if ((trityp==3)&&(j+k>i)) trityp = 2 ;
26            else trityp = 4 ; // not a triangle
27          }
28      }
29    return trityp;
30  }
```

FIG. III.14 – Programme Tritype de classification d'un triangle

CBMC n'a pas permis de trouver l'erreur. Nous avons contacté D. Kroening qui a recommandé d'utiliser l'option CPROVER_assert. Avec cette option, CBMC est capable de trouver l'erreur dans la version erronée de Tritype. Cependant, en utilisant la même option pour la version correcte du programme, nous devons également ajouter la pré-condition $CPROVER\_assume(i+j >= 0 \&\& i+k >= 0 \&\& j+k >= 0)$ afin d'assurer qu'il n'y a pas de débordement dans la somme.

Un tri à bulles avec des conditions initiales Cet exemple est extrait de [4] et effectue un tri à bulles d'un tableau t qui contient les entiers de 0 à $t.length$ classés en ordre décroissant. Sur cet exemple, CPBPV est moins efficace que CBMC : il ne peut pas traiter un tableau de longueur 32 alors que CBMC le traite en 6.10s. Ceci provient des nombreux tests de consistance qui sont effectués, et surtout de l'implémentation récursive qui consomme toute la pile d'exécution Java ; il faudrait donc dé-récursiver l'implémentation. Notons que cet exemple est par contre résolu efficacement avec notre première approche (voir section III.3.4 page 109).

Tri par sélection Cet exemple illustre l'usage de la contrainte *element* (voir section III.2.2). En effet, contrairement au cas du tri à bulles où les deux boucles emboîtées fixent les indices d'accès au tableau, l'indice du plus petit élément de la partie non triée du tableau est inconnu dans le cas du tri par sélection. Ainsi, la contrainte *element* est nécessaire pour exprimer les affectations mises en jeu lors de l'échange de l'élément d'indice i courant et de l'élément minimum.

Nous avons effectué une vérification modulaire du tri par sélection en vérifiant la fonction `findMin` qui renvoie l'indice de l'élément minimum des éléments du tableau d'indices compris entre 0 et i. La vérification de la fonction de tri par insertion explore l'unique chemin en moins de $0.01s$ pour un tableau de longueur 40. La difficulté réside dans la vérification de la fonction `findMin` qui explore $2^{t.length}$ chemins. Cependant, la vérification complète du tri par sélection a pris moins de $4s$ pour un tableau de longueur 6, ce qui est satisfaisant par rapport à d'autres approches par bounded model checking. Sur une version où les entrées sont fixées comme dans le tri à bulles (i.e. valeurs entre 0 et $t.length$ par ordre décroissant), EUREKA met 104s sur une machine plus rapide [4]. De plus, il est indiqué dans [4] que CBMC prend $432.6s$, que BLAST ne peut pas résoudre ce problème, et que SATABS [15] peut le faire seulement pour un tableau de deux éléments.

Somme des carrés des n premiers entiers Ce dernier exemple calcule la somme des carrés des n premiers entiers qui sont contenus dans un tableau. Il illustre le pouvoir d'expression et de résolution spécifique aux contraintes, en particulier de la contrainte globale $all - diff$ [39]. En effet, la pré-condition exprime que le tableau contient des valeurs comprises entre 0 et $t.length - 1$ qui sont toutes différentes ; il s'agit donc d'une permutation quelconque de ces valeurs. L'instance maximale que nous avons pu résoudre avec CPBPV est un tableau de taille 10 en $66.179s$.

Discussion

Le solveur linéaire joue un rôle majeur dans tous ces exemples, puisqu'ils contiennent de nombreuses expressions linéaires. Par exemple, le solveur sur domaines finis (solveur CP par la suite) n'est jamais appelé pour Tritype ; ici, toutes les expressions sont linéaires et le solveur linéaire permet à la fois de montrer la faisabilité des chemins et l'inconsistance du système final.

Pour la recherche binaire dans un tableau t, le solveur CP consomme 75% du temps CPU pour résoudre la contrainte `element`. En effet, pour le cas de spécification où l'élément est trouvé dans le tableau (i.e. $result != -1$ et $t[result] = x$), les valeurs t, $result$ et x sont quelconques dans la contrainte `element` qui représente l'égalité $t[result] = x$. La propagation est donc coûteuse.

Puisque le tri à bulles explore un unique chemin, il y a un seul appel au solveur CP à la fin de ce chemin pour traiter les contraintes `element` de la spécification (les contraintes `element` du programme sont résolues puisque tous les indices et valeurs sont connues).

Dans l'exemple de la somme des carrés des n premiers entiers, 80% du temps CPU est passé dans le solveur CP puisque la spécification aussi bien que le programme contiennent des expressions non linéaires.

III.5 Conclusion

Les deux approches que nous avons proposées utilisent la programmation par contraintes pour effectuer une vérification partielle incomplète de triplets de Hoare. Les résultats expérimentaux ont montré l'efficacité et la simplicité d'utilisation de ces approches. L'usage de la programmation par contraintes pour ce problème est novateur. En effet, la programmation par contraintes a déjà été utilisée pour la génération de jeux de tests [29, 32, 41] et a fourni un outil d'implémentation des techniques d'exécution symbolique pour les programmes en flottant [7]. Flanagan [26] a formalisé la traduction des programmes impératifs dans le cadre CLP (Constraint Logic Programming), et a montré que cela pouvait servir pour le bounded model checking, mais il n'a pas réalisé d'implémentation.

Dons notre étude expérimentale, nous avons évalué des catégories d'outils basés sur des techniques différentes : l'analyse statique abstraite, la génération de plus faibles pré-conditions, le model checking et le bounded model checking.

L'analyse statique procède par propagation des valeurs qui peuvent être prises par les variables dans les différentes branches du programme jusqu'à atteindre un point fixe. L'analyse statique abstraite procède de plus en utilisant une abstraction des domaines. L'outil d'analyse statique ESC/Java [16, 12] que nous avons évalué nécessite l'introduction d'annotations détaillées pour réduire les informations que l'analyseur doit effectuer. Ceci complique donc la tâche du programmeur. De même, les méthodes complètes qui génèrent des plus faibles pré-conditions [25, 2, 11] nécessitent un effort supplémentaire pour fournir des invariants de boucle. L'avantage est bien évidemment que la vérification est *complète*.

Sur les exemples que nous avons traités, CPBPV est plus efficace que BLAST, un outil de model checking qui effectue une abstraction paresseuse [7]. L'atout de

CPBPV par rapport au model checking est qu'il ne génère pas de contre-exemples fallacieux et qu'il n'explore pas les chemins qui sont trivialement infaisables. Notre première approche par abstraction booléenne donne aussi de meilleurs résultats sur certains exemples cités dans [5], puisque nous traitons au sein du même solveur à la fois l'abstraction booléenne du contrôle mais aussi les contraintes numériques de la partie opérative. Nous générons donc moins de contre-exemples fallacieux. Il est à noter cependant que les outils basés sur le model checking avec abstraction et raffinement [13, 14] sont en général dédiés à la vérification de propriétés spécifiques, telles que les propriétés de sûreté qui correspondent à un problème d'atteignabilité dans le graphe de transition d'états. Vérifier de telles propriétés ne nécessite pas toujours, contrairement à la vérification d'un triplet de Hoare, d'explorer tous les chemins. Le model checking semble donc mieux adapté à ce type de propriétés.

Nos deux approches sont similaires aux approches de bounded model checking qui utilisent un solveur SAT comme CBMC [13, 14] et F-Soft [31] ou qui utilisent un solveur SMT comme l'approche proposée par Armando dans [5, 4, 6]. Notre première approche par abstraction booléenne est très proche des mécanismes mis en jeu dans les solveurs SMT paresseux puisque la théorie des entiers est utilisée a posteriori pour vérifier si la solution booléenne trouvée est valide quand on la concrétise dans les entiers. Il y a toutefois deux différences majeures. D'une part, les solutions booléennes ne concernent que les conditions de la partie contrôle ; l'abstraction est donc guidée par le flot de contrôle du programme. D'autre part, nous manipulons à la fois les booléens et les entiers et la propagation inverse des contraintes gardées permet de réduire l'espace des solutions des variables booléennes. La comparaison de CPBPV avec CBMC et avec les résultats expérimentaux publiés dans [5] ont démontré son efficacité. Cela est dû principalement à la performance du solveur linéaire. Un atout supplémentaire de CPBPV par rapport aux autres approches est qu'il peut aussi traiter des contraintes non-linéaires, ainsi que des accès à des tableaux quand les valeurs des indices sont inconnues grâce à la contrainte *element*. L'algorithme d'exécution symbolique est également très similaire à celui présenté dans [1] qui utilise la librairie Omega [44] et génère automatiquement certains invariants de boucle. Nous n'avons pas pu comparer de façon expérimentale CPBPV à cette approche, mais la génération des invariants est une voie qui nous paraît intéressante à explorer.

CPBPV a par contre montré des limites dans l'exploration du graphe de flot de contrôle du programme. Le problème est d'une part l'exploration récursive (l'implémentation pourrait en être améliorée) et d'autre part le fait que l'algorithme d'exécution symbolique teste toutes les conditions des instructions de contrôle. Même si cela permet de couper certains chemins, cela a un coût non négligeable, surtout dans l'implémentation actuelle où le solveur est appelé de façon naïve depuis Java. Nous verrons que Le Vinh Nguyen a proposé une nouvelle implémentation en COMET [43] qui semble plus efficace car elle tire parti du non déterminisme du langage et de la gestion incrémentale du système de contraintes. Ceci sera présenté brièvement dans les perspectives (voir chapitre VI section VI.1.3 page 262). D'autre part, nous discutons en détail dans le chapitre suivant d'autres méthodes pour générer les formules issues de la spécification et du programme en utilisant HOL4 et un solveur SMT.

III.6 Articles supports

III.6.1 Exploration of the capabilities of Constraint Programming Techniques for Software Verification

H.Collavizza, M. Rueher. Exploration of the capabilities of Constraint Programming Techniques for Software Verification. TACAS'06, LNCS 3020, Vienne, Mars 2006, pp 182-196.

Exploration of the capabilities of constraint programming for software verification*

Hélène Collavizza and Michel Rueher

Université de Nice–Sophia-Antipolis – I3S/CNRS
930, route des Colles - B.P. 145, 06903 Sophia-Antipolis, France.
{helen,rueher}@essi.fr

Abstract. Verification and validation are two of the most critical issues in the software engineering process. Numerous techniques ranging from formal proofs to testing methods have been used during the last years to verify the conformity of a program with its specification. Recently, constraint programming techniques have been used to generate test data. In this paper we investigate the capabilities of constraint programming techniques to verify the conformity of a program with its specification. We introduce here a new approach based on a transformation of both the program and its specification in a constraint system. To establish the conformity we demonstrate that the union of the constraint system derived from the program and the negation of the constraint system derived from its specification is inconsistent (for the considered domains of values). This verification process consists of three steps. First, we generate a Boolean constraint system which captures the information provided by the control flow graph. Then, we use a SAT solver to solve the Boolean constraint system. Finally, for each Boolean solution we build a new constraint system over finite domains and solve it. The latter system captures the operational part of the program and the specification. Boolean constraints play an essential role since they drastically reduce the search space before the search and enumeration processes start. Moreover, in the case where the program is not conforming with its specification, Boolean constraints provide a powerful tool for finding wrong behaviours in different execution paths of the program. First experimental results on standard benchmarks are very promising.

1 Introduction

Verification and validation are two of the most critical issues in the software engineering process. These expensive and difficult tasks may account for up to 50% of the cost of software development [12]. Numerous techniques ranging from formal proofs to testing methods have been used during the last years to verify the conformity of a program with its specification. The goal of the SLAM project is to build "tools that can do actual proofs about the software and how it works in order to guarantee the reliability"[1].

* This research was partially supported by the RNTL project "DANOCOPS".

[1] see http://research.microsoft.com/slam

Constraint programming techniques have been used to generate test data (e.g., [9, 10, 22, 23]) and to develop efficient model checking tools (e.g. [17, 6]). SAT based model checking platforms have been able to scale and perform well due to many advances in SAT solvers [20]. Recently Bouquet et al [3] developed a symbolic animator for specifications written in Java Modeling Language (JML) [15]. Their JML animator– based on constraint programming techniques –allows to simulate the execution of a JML specification and to verify on the fly class invariant properties.

In this paper we investigate the capabilities of constraint programming techniques to verify the conformity of a program with its specification. We introduce a new approach based on a transformation of both a program and its specification in a constraint system. To establish the conformity we demonstrate that the union of the constraints derived from the program and the negation of the constraints derived from its specification is inconsistent. Roughly speaking, pruning techniques -that reduce the domain of the variables- are combined with search and enumeration heuristics to demonstrate that this constraint system has no solutions.
The verification process consists of three steps:

1. Generating of a Boolean constraint system which captures the information provided by the control flow graph of the program and the specification;
2. Using a SAT solver to find the solutions of the Boolean constraint system. For each Boolean solution a new constraint system over finite domains – denoted CSP in the following– is built; the latter captures the operational part of the program and the specification.
3. Solving the CSP with a finite domain solver.

Boolean constraints play an essential role since they drastically reduce the search space before the search and enumeration processes start on the generated CSP. Moreover, in the case where the program is not conforming with its specification, Boolean constraints provide a powerful tool for finding wrong behaviors in different execution paths of the program. An essential observation is that in this approach we do not transform all assignments and numerical instructions into Boolean constraints[2]. The point is that it is much more convenient to transform these instructions in finite domain constraints and to solve them with a CSP solver. So the collaboration between the SAT solver and the CSP solver is the cornerstone of our approach. Indeed, since we first identify the feasible paths, the finite domain solver will work with both smaller constraint systems and reduced domains.

The prototype system we have developed takes as input a JAVA program and its specification written in JML [15]. Currently, we only consider JAVA unit code without function calls, without return inside loops, and without inheritance. Moreover, we assume that all numerical operations only concern integers.

[2] Contrary to the most popular Model Checking approaches based on SAT Solvers [4, 5].

The rest of this paper is organised as follows. Section 2 gives an overview of our approach whereas Section 3 recalls some basics on constraint programming techniques. Section 4 details the verification process we propose and introduces the translation process we use to generate the constraint systems. Section 5 describes the experimental results and discusses some critical issues.

Before going into the details, let us illustrate the capabilities of our approach on a well-known benchmark.

2 Motivating Example

We illustrate our approach on the well-known `tritype` program for classification of triangles [7]. We first describe the program, then we show very informally how the transformation process works, and finally we describe different experimentations.

2.1 The Problem

The `tritype` program is a basic benchmark in test case generation since it contains numerous non feasible paths. `tritype` takes three positive integers as inputs (the triangle sides) and returns 1 if the inputs correspond to any triangle, 2 if the inputs correspond to an isoscele triangle, 3 if the inputs correspond to an equilateral one, 4 if the inputs do not correspond to any triangle. Figure 1 gives the `tritype` program in JAVA with its specification in JML. Note that `\result` in JML corresponds to the value returned by the program.

2.2 The verification process

We first translate this program and the negation of its specification into a set of constraints, using the process detailed in Section 4.2.

Then, in order to delay the enumeration on integers, we introduce boolean variables for each decision on input variables, e.g., we introduce variable eq_{ij} for condition $i = j$, variable eq_{ik} for condition $i = k$, and so on. So the resolution process is decomposed into two parts:

1. Finding a set of paths that correspond to potential non-conformities;
2. Solving the CSP which corresponds to the identified set of paths.

For instance, if the boolean solver finds the solution $\{eq_{ij} = true, eq_{jk} = true, eq_{ik} = false\}$ we generate the CSP $\{i = j, j = k, i \neq k\}$; the domain of i, j, k being {0,...,65635}. If the CSP has a solution we have found a test case which corresponds to a non-conformity. If none of the generated CSP has a solution the verification is done.

The constraints generated for lines 4 to 7 in Fig. 1 are displayed in Fig. 2. $cond \rightarrow c$ denotes a guarded constraint: roughly speaking, constraint c has to be satisfied when condition *cond* holds (see section 3.3 for the exact semantic).

```
/*@ public normal_behavior
  @ requires (i>=0)&&(j>=0)&&(k>=0);
  @ ensures
  @   ((i+j<=k)||(j+k<=i)||(i+k<=j)) ==> \result == 4 &&
  @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&((i==j)&&(j==k)) ==> \result == 3 &&
  @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
  @        &&((i==j)||(j==k)||(i==k)) ==> \result == 2 &&
  @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
  @        &&!((i==j)||(j==k)||(i==k)) ==> \result == 1;
@*/

1  public static int tritype(int i, int j, int k){
2   int trityp ;
3   // not a triangle
4   if ((i==0)||(j==0)||(k==0)) trityp = 4 ; //ERR: trityp = 3
5   else {
6     trityp = 0 ;
7     if (i==j) trityp = trityp + 1 ;
8     if (i==k) trityp = trityp + 2 ;
9     if (j==k) trityp = trityp + 3 ;
10    if (trityp==0){
11       // triangular inequality not verified
12       if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4 ;
13       else trityp = 1 ; // any triangle
14    }
15    else {
16      if (trityp > 3) trityp = 3 ; // equilateral
17      else
18        //i=j and triangular inequality verified
19        if ((trityp==1)&&(i+j>k)) trityp = 2 ;
20        else
21          //i=k and triangular inequality verified
22          if ((trityp==2)&&(i+k>j)) trityp = 2 ; //ERR: (trityp == 1)
23          else
24            //j=k and triangular inequality verified
25            if ((trityp==3)&&(j+k>i)) trityp = 2 ;
26            else trityp = 4 ; // not a triangle
27          }
28      }
29    return trityp;
30  }
```

Fig. 1. `tritype` program in java with a specification in JML

$r0$ and $r1$ are the two first renamings of variable r. nul_i (resp. nul_j, nul_k) is the boolean variable that captures the decision $i = 0$ (resp. $j = 0$, $k = 0$). eq_{ij} is the boolean variable for decision $i == j$. The constraint $((nul_i = 1) \vee (nul_j = 1) \vee (nul_k = 1)) \rightarrow r0 = 4$ corresponds to the `if` part of instruction on line 4 in Fig. 1, the following constraint corresponds to the else part. The last two constraints correspond to the `if` instruction on line 7 in Fig. 1. The full constraint system for the `tritype` program can be found in http://www.essi.fr/ rueher/appendix-tacas06.pdf.

// SSA variables for multiple definitions of result in the program	
r0 : {0,...,65635}, r1 : {0,...,65635},	
// boolean variables	
nul_i : {0,1}, nul_j : {0,1}, nul_k : {0,1}, eq_{ij} : {0,1}	
//constraints of line 4 to 7 of the program	
$((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1))$	$\rightarrow$ r0=4
$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1))$	$\rightarrow$ r0=0
$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \wedge (eq_{ij}{=}1)$	$\rightarrow$ r1=r0 + 1
$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \wedge \neg\ (eq_{ij}{=}1)$	$\rightarrow$ r1=r0

Fig. 2. Constraints generated for lines 4 to 7 of the `tritype` program

2.3 Experimentations

We have introduced two errors into the `tritype` program:

1. A wrong return value when one of the inputs is zero (line 4 of the java program);
2. A wrong test on the trityp variable (line 22 of the java program).

These two errors occur in two different execution paths of the program. Figure 3 displays the four first non-conformities we have found: we successively display the path (i.e the value of decision variables), then three solutions of the corresponding integer system, and finally the value returned by the specification and the program.
The two first non-conformities are due to the wrong test on variable *trityp*, line 22 of the program. The first one is generated when "i=k", and so "trityp=2". Since the test on line 22 is "trityp==1" instead of "trityp==2", the execution goes through the else part on line 25, so the value of result equals 4 instead of 2. The second non-conformity corresponds to the case where "i=j". So "trityp=1" and due to the wrong test on line 22 result equals 2 instead of 4 since the triangular inequality is verified.
The other errors we have found are those where at least one of the input is zero. Since we have introduced the error "trytyp = 3" instead of "trytyp = 4" on line

4 of Fig. 1, the program returns 3 instead of 4 whenever an input is equal to zero. The overall process finds 15 non-conformities in less than 5 seconds CPU time[3]

We did also run the verification process with a correct program. It required 2.36 seconds CPU time to perform the complete verification. Note that we explored only 92 solutions of the Boolean constraint system although there are 9 variables, and thus 2^9 combinations. This clearly shows that the constraint system is strong enough to prune the search space, and to avoid a costly enumeration of all paths.

Error 1
Path : !(i=j), i=k, !(j=k), !(i+j≤ k), !(j+k ≤ i), !(i+k ≤ j), !(i=0), !(j=0), !(k=0)
Input values : i:2, j:1, k:2 – i:2, j:3, k:2 – i:3, j:1, k:3
Specification : 2, program : 4
Error 2
Path :i=j, !(i=k), !(j=k), i+j ≤ k , !(j+k ≤ i), !(i+k ≤ j), !(i=0), !(j=0), !(k=0)
Input values : i:1, j:1, k:2 – i:1, j:1, k:3 – i:1, j:1, k:4
Specification : 4, program : 2,
Error 3
Path : !(i=j), !(i=k), !(j=k), !(i+j ≤ k) , !(j+k ≤ i), i+k ≤ j, !(i=0), !(j=0), k=0
Input values : i:1, j:2, k:0 – i:1, j:3, k:0 – i:1, j:4, k:0
Specification : 4, program : 3
Error 4
Path : !(i=j), !(i=k), !(j=k), !(i+j ≤ k), j+k ≤ i, !(i+k ≤ j), !(i=0), !(j=0), k=0
Input values: i:2, j:1, k:0 – i:3, j:1, k:0 – i:3, j:2, k:0
Specification : 4, program : 3

Fig. 3. Four first non-conformities for the `tritype` program with two errors

3 Constraint programming

This section recalls some basic concept of constraint programming which are useful to understand this paper. More details can be found in [21, 18, 13].

3.1 Definition of a CSP

Constraint programming is a paradigm that is tailored to hard search problems. The main application areas are planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming

[3] All experimentations have been performed with ILOG Solver (see http://www.ilog.com/products/solver) and run on a Intel(R) Pentium(R) 4 CPU 2.00GHz computer with 256 Mb memory.

incorporates techniques from mathematics, artificial intelligence and operational research; it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance.

Constraint programming solvers are based on a *branch and prune* algorithm that combines local consistencies and efficient search heuristics.

More precisely, a *Constraint Satisfaction Problem* (CSP) is defined as:

- a set of *variables* $X = \{x_1, ..., x_n\}$,
- a finite set D_i of possible values for each variable x_i, called *domain*,
- a set of *constraints* $C = \{c_1, ..., c_n\}$ restricting the values that the variables can simultaneously take; X_j denotes the set of variables that occur in constraint c_j.

Note that the domains are a convenient way to express some specific constraints.

A *solution of a CSP* is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied.

3.2 Solving a CSP

To solve a CSP pruning techniques -that reduce the domain of the variables- are combined with search and enumeration heuristics. We only detail here local consistencies techniques.

Local consistencies are a key issue in finite domains where arc–consistency [19, 16] is very popular. A constraint c_j is arc-consistent if for any variable x_i in X_j, each value in D_i has a support in the domains of all other variables of X_j. In other words, a constraint c is arc–consistent for variable x, if values exist in the domains of all other variables such that constraint c holds when x is assigned to any value of its domain. The essential observation is that local consistency filtering algorithms try to reduce the size of the domain of some variable by considering only one constraint.

The following example shows in a very informal way how arc–consistency works. Consider the constraint system $C_1 = \{c_1 : x_1 + x_2 > 2, c_2 : x_1^2 + x_2^2 \leq 4, D_1 = \{0, 1, 2\}, D_2 = \{0, 1, 2\}\}$.
Constraint c_1 cannot be satisfied when either x_1 or x_2 are equal to 0. So arc–consistency will remove value 0 from domain D_1 and domain D_2. Now, constraint c_2 can no longer be satisfied when x_1 or x_2 are equal to 2, and thus value 2 will be removed from both domains. However, since the domain of one of the variables of constraint c_1 has been modified, we have to reconsider this constraint. Now, c_1 can no more be satisfied, the value 1 is removed from its domain which become empty; thus arc-consistency has detected the inconsistency of the whole constraint system.
Constraint system C_2 (see below) shows a case where the constraint system is arc-consistent but no solution satisfying all constraints exists.
$C_2 = \{c_1 : x_1 \neq x_2, c_2 : x_2 \neq x_3, c_1 : x_3 \neq x_2\}, D_1 = D_2 = D_3 = \{0, 1\}$.

3.3 Guarded constraints

In this paper we also use guarded constraints. Guarded constraints are conditional constraints whose evaluation depends upon other constraints. $C_0 \rightarrow C_1$ denotes a guarded constraint where C_0 and C_1 are conjunctions of basic constraints. Relation $C_0 \rightarrow C_1$ states that constraints C_1 have to be added to the current constraint store when the solver can prove that constraints C_0 hold. More precisely, let C_0 be a boolean expression and C_1 a set of constraints, the guarded constraint $C_0 \rightarrow C_1$ behaves as follows:

- When the solver can prove that C_0 is true, then constraints C_1 are added to the store of constraints;
- When the solver can prove that C_0 is false, then the guarded constraint is just discarded;
- When the solver can neither prove that C_0 is true, nor prove that C_0 is false, that is when not enough variables of C_0 are instantiated, then the guarded constraint is suspended.

The solver tries to prove that the guard C_0 of a suspended constraint holds whenever the domain of some variable occurring in C_0 has been reduced. Of course, some guarded constraints may never become active.

One major difficulty with guarded constraints is that nothing can be done before the solver can demonstrate that the condition is either *true* or $false$. Let us consider a very simple piece of code:

```
//@ ensures \result ≥ 0
public int absolute(int i, int j) {
   if (i<j) return j-i;
        else return i - j;
}
```

This code is translated into the following set of constraints:

$\{i < j \rightarrow r = j - i, \neg(i < j) \rightarrow r = i - j, r < 0, D_i = D_j = D_r = \{0, ..., 65635\}\}$

A standard CSP solver cannot achieve any pruning on this system since nothing is known about i and j. So a very costly enumeration process is started: the inconsistency is only detected when the domain of i and j are reduced to one value. The advantages of combining SAT solver and CSP are obvious here. After having introduced a boolean variable for modeling $i < j$, the SAT solver enumerates the two paths, that is to say the two CSP $\{r = j - i, i < j, r < 0\}$ and $\{r = i - j, i \geq j, r < 0\}$. When the constraints of the CSP are transformed in binary constraints, arc-consistency immediately detects the inconsistency.

4 Verification process

In this section we describe the overall verification process and explain how we transform the program and its specification into a set of constraints.

4.1 Verification steps

The different operations which are performed during the verification process are detailed in Fig. 4.

1. Put the program into a simplified Single State Assignment (SSA) form [14] and translate the SSA program into a set of constraints.
2. Add the constraints corresponding to the negation of the property to be proved.
3. Introduce a boolean variable for each decision on an input variable; Let *BoolSystem* be the constraint system obtained after steps 1, 2, and 3.
4. Start a solving process on *BoolSystem* and for `each` solution of *BoolSystem*:
   a. Build a CSP *IntSystem* that corresponds to the boolean values found in the current solution of *BoolSystem*
   b. Start a solving on IntSystem and for `each` solution of IntSystem print the current values of boolean variables (path trace) and find some errors of *IntSystem* (wrong input values).
5. If *BoolSystem* has no solution or if for each boolean solution *IntSystem* has no solution, print "the program is conform with is specification".

Fig. 4. verification process

Note that in step 3, we introduce boolean variables only to model decisions about input variables. This is sufficient to delay the enumeration process induced by guarded constraints (see Section 3.3). On the other hand, assignments are modeled using integer variables. Thus, we lose less information than with a translation of any statement into a boolean variable.

4.2 Translating the program into a set of constraints

We first transform the program into its SSA form: for each new definition of a program variable, we introduce a fresh variable. In order to manage control instructions, we use ϕ–functions for `if then else` statements and we unfold loops. We use guarded constraints to model conditional execution flow (see part 3.3).

Basic statements Each assignment $var \leftarrow value$ is translated as a constraint $var = value$. Each boolean condition is translated as the corresponding constraint. We denote $SSA(s)$ the constraint corresponding to the basic statement s where each new definition of a variable has been replaced by the current renaming of this variable.

The `If then` statement For the sake of clarity, we only focus on the assignment of a single variable. Trivially, the same process could be applied individually for

each variable appearing in a block with many variable assignments. Let us consider the statement `S : if (cond)` {`var=val1;var=val2; ...; var=valq;`}. Assume that *var* has already been defined p times before this statement. S is translated into the following set of guarded constraints:

$$
\begin{array}{lcl}
\text{SSA(cond)} & \rightarrow & var_{p+1} = \text{SSA}(val_1) \\
\text{SSA(cond)} & \rightarrow & var_{p+2} = \text{SSA}(val_2) \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q} = \text{SSA(valq)} \\
// \text{ else part} & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+1} = var_p \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+2} = var_p \\
\ldots & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+q} = var_p
\end{array}
$$

The else part is useful to ensure that the q fresh variables will not remain uninstantiated in the corresponding CSP.

The `If then else` statement Let us consider the statement `S : if (cond)` {`var=val11;var=val12; ...;var=val1q;`} `else` {`var=val21;var=val22;... var=val2r;`}. Assume that *var* has already been defined p times before this statement and assume that $q < r$. Since *var* has not the same number of definitions in the `if` part and the `else` part, we need to introduce a guarded constraint to take the place of the ϕ function. So, S is translated into the following set of guarded constraints:

$$
\begin{array}{lcl}
// \text{ if part} & & \\
\text{SSA(cond)} & \rightarrow & var_{p+1} = \text{SSA}(val_{11}) \\
\text{SSA(cond)} & \rightarrow & var_{p+2} = \text{SSA}(val_{12}) \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q} = \text{SSA}(val_{1q}) \\
// \text{ else part} & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+1} = \text{SSA}(val_{21}) \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+2} = \text{SSA}(val_{22}) \\
\ldots & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+r} = \text{SSA}(val_{2q}) \\
// \ \phi \text{ function} & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q+1} = var_{p+q} \\
\text{SSA(cond)} & \rightarrow & var_{p+q+2} = var_{p+q} \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+r} = var_{p+q}
\end{array}
$$

Remark: if $q > r$ the same principle is applied and the guarded constraints of the ϕ function are guarded by $SSA(\neg\ cond)$. If q=r then no ϕ function is required.

Figure 5 gives the translation of an overlapped `if then else`.

```
1  if (i < j) x = 0;                 (i<j) → x1=0
     else {                          (¬(i<j)∧(i<30))→ (x1=x0+1∧x2=x1+y0)
2       if (i < 30) {                (¬(i<j)∧ ¬(i<30)∧(j>43))→ x1=2
           x = x+1;                  (¬(i<j)∧ ¬(i<30)∧ ¬(j>43))→ x1=3
           x = x+y;                  // φ-function for #2 if
        }                            (¬(i<j)∧¬(i<30)) → x2=x1
        else {                       // φ-function for #1 if
3          if (j > 43) x=2;          (i<j) → x2=x1
           else x=3;
        }
     }
```

Fig. 5. example of if then else translation

The `Loop` statement We first transform any loop into the equivalent while loop. Then we unfold the while loop using an overestimate of the number of loop steps. This overestimate may be the worst case complexity of the loop or could be given by the user. To describe all possible paths inside the loop, we guard the constraints of the loop with the entrance condition. Our process is close to the one described in [4] except that we use guarded constraints instead of boolean operators to combine condition and assignment. More precisely, let us consider the following while loop `L : while (cond) var = value;`. We assume that this loop is executed at most max time and that var was defined p times before this statement. Then the loop statement L is translated into the following set of guarded constraints:

$$
\begin{array}{ll}
cond_1 & \rightarrow var_{p+1} = val_{p+1} \\
\neg\ cond_1 & \rightarrow var_{p+1} = var_p \\
cond_1 \wedge\ cond_2 & \rightarrow var_{p+2} = val_{p+2} \\
\neg\ (cond_1 \wedge\ cond_2) & \rightarrow var_{p+2} = var_{p+1} \\
..... & \\
cond_1 \wedge\ cond_2 \wedge\ ... \wedge\ cond_{max} & \rightarrow var_{p+max} = val_{p+max} \\
\neg\ (cond_1 \wedge\ cond_2 \wedge\ ... \wedge\ cond_{max}) & \rightarrow var_{p+max} = var_{p+max-1}
\end{array}
$$

where val_i denotes the ith SSA form of `value`.

With this system of guarded constraints, if the loop condition has never been true, then $var_{p+max} = var_p$, if it has been true only once then $var_{p+max} = var_{p+1} = val_{p+1}$, if it has been true max times then $var_{p+max} = val_{max}$.

Remarks:

- The number of unfoldings may not be sufficient to detect all non-conformities especially when an error in the program entails more iterations than specified by the theoretical bound.

- When the bound of a for loop is well-known and when the index variable is not modified inside the loop block, it is more efficient to generate n constraint systems, one for each value of the decision variable. This is due to the fact that

the guarded constraints are expensive to manage, even when the conditions are instantiated very early.

5 Experimental results and discussion

In this section we analyse the experimentations we have performed on three non-trivial academic examples.

5.1 The `tritype` program

The first example we consider is the `tritype` program. As we mentioned in the introduction, we can find some errors in the program as well as prove the correctness of the program. Introducing boolean variables only for decisions on input variables gave very good results in this case. Indeed, `tritype` is a typical example of pure decisional program, so the proof mainly consists in showing that the same decision in program and specification gives the same code condition return value.

5.2 The `merge` example

This `merge` program referenced in [11] computes five outputs from five inputs[4]. A partial order is given on the inputs and the property to be proved is that the outputs are sorted in decreasing order. In this program, a contrario to tritype program, the link between the specification and the program goes through the operational part. So we need to introduce boolean variables not only to model decisions but also to model the assignments. We have introduced the same error as in [11]. We found four error paths including the one shown in [11]. For each error path we search five different integer values for the inputs. The overall process took 159.71 seconds CPU time whereas the proof of the correctness required 310.67 seconds CPU time (no CPU time is given in [11]).

5.3 The `bsearch` program)

The `bsearch` program[5] takes as input an array of integers t sorted by increasing order, an integer value val to search in the array, and returns the index of the value if it is found or -1 otherwise. The worst case complexity of this program is $O(log_2(n))$ where n is the size of the array.

To perform the verification, we introduce boolean variables for condition tests on input variables (i.e $t[i] = val$, $t[i] < val$, $t[i] < t[i+1]$). Since the worst case complexity of bsearch is $O(log_2(n))$ we unfold the program loop $\lceil(log_2(n))\rceil$

[4] The `merge` program and its JML specification can be found in http://www.essi.fr/ rueher/appendix-tacas06.pdf

[5] The `bsearch` program and its JML specification can be found in http://www.essi.fr/ rueher/appendix-tacas06.pdf

times. We successively introduce two errors. The first one is to return the value $middle+1$ when the $t[middle] = val$. This error was detected by the CSP solver. The errors found by the solver correspond to all the possible paths through the loop when it stops with $t[middle] = val$. The second error consists in assigning the right bound with $middle$ instead of $middle + 1$ when $t[middle] < val$. With this second error the program will not terminate in some cases, for example when searching a value which is bigger than all the values in the array. This error was also detected.

The correctness proof was also performed. The required time exponentially increases according to the length of the array. The solver runs out of memory for arrays of size > 8 and values in $[0,2^{16}]$.

6 Discussion and related work

The new framework we have introduced in this paper has of course some limitations, e.g., there is no way to prove temporal properties, it works well with JAVA program but it would be difficult to handle C programs with pointers. Even for JAVA programs there are some restrictions: inheritance and functions calls are currently not handled[6].

A critical issue concerns the detection of inconsistencies in the CSP generated for each Boolean constraint system. Indeed, the constraints in some CSP may be too weak to achieve any pruning of the solution space, even when this CSP has no solution. In this case, a very costly search process is required to demonstrate that the CSP is inconsistent. To overcome - at least partly - this problem some dedicated solvers could be used. For instance, when the finite domain constraints are linear, linear programming solvers could be used to reduce the domains. Formal simplifications of the constraint system could also be useful in some cases.

Of course, this problem is highly dependent from the modelling of the program and its specification. In other words, the kind of constraints that are generated will have strong influence on the performances of the solver [7].

Ganziger et al [8] have introduced a general DPLL(X) engine, where parameter X can be instantiated with a specialized solver $Solver_T$. That's to say DPLL(X) is a general engine for propositional solving. The authors illustrate their approach on their solver for EUF(logic with equality with uninterpreted functions). The goal of the approach introduced in this paper is not to integrate a CSP solver in a general DPLL engine. In our framework the essential role of the SAT solver is to boost the CSP solver by reducing the search space.

Armando et al [1] have recently proposed to use SMT solvers instead of SAT solvers for bounded model checking of software. We have compared our solver with their SMT-CBMC solver, which use CVC Lite for the theory of bit vectors.

[6] As long as we only consider finite structures, it should be possible to incorporate these features into our framework without major difficulties

[7] This is a well known problem in constraint programming: the performances of a solver may be very different on two constraint systems that are logically equivalent.

We have performed experimentations on the two sorting benchmarks contained in their last paper [1]. SMT-CBMC requires more than 600 seconds to analyse a bubble sort program with an array of size 26 whereas our solver analyses the same program with an array of size 100 in less than one second. Similarly, SMT-CBMC requires more about 200 seconds to analyse a selection sort program with an array of size 29 whereas our solver analyses the same program with an array of size 100 in less than 3 seconds.

We have also started an evaluation of our framework on standard SMT benchmarks (http://www.csl.sri.com/users/demoura/smt-comp/2005/). First results are promising; for instance we did prove the unsatisfiability of DTP_k2_n35_c210-s7.smt and DTP_k2_n35_c245_s10.smt in less than one second.

7 Conclusion

In this paper we have performed a first exploration of the capabilities of constraint techniques for verifying the conformity of a program with its specification.

First experimentations show that these techniques can be very efficient on some non trivial problems. Further work concerns the inclusion of dedicated solvers or simplifiers in our framework as well as a deeper study of the modelling issue.

Acknowledgements: Many thanks to Laurent ARDITI and Claude MICHEL for numerous and enriching discussions on this work.

References

1. Armando, A., Mantovani, J., and Platania, L.: Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver Technical Report, AI-Lab, DIST, University of Genova, December 19, 2005, 16 pages.
2. Ball T., Rajamani S. K., : Boolean Programs : A Model and Process For Software Analysis. Technical Report MSR TR 200-14, 2000
3. Bouquet, F., Dadeau, F., Legeard, B. and Utting, M: JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440,pp. 551–556, 2005.
4. Clarke E., Kroenig D., Lerda F. : A Tool for Checking ANSI-C programs. TACAS 2004, LNCS 2988, pp 168-176, 2004
5. Clarke E., Kroenig D., Sharygina N., Yorav K. : Predicate abstraction of ANSI-C Programs using SAT. Formal Methods in System Design, Vol **25**, pp 105-127, Kluwer Academic Press, 2004
6. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav: SATABS: SAT-Based Predicate Abstraction for ANSI-C. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440,pp. 570–574, 2005.
7. Demillo R. A., Offut A.J. : Experimental Results from an Automatic Test Case Generator. ACM Transactions on Software Engineering Methodology. vol. **2**, number 2, 1993, pp 109-175

8. Ganzinger,H., Hagen,G., Nieuwenhuis, R.,Oliveras, A., and C. Tinelli: DPLL(T): Fast Decision Procedures. Proc. of CAV 2004. Springer-Verlag. LNCS 3114,pp. 175-188, 2004.
9. Gotlieb A., Botella B. and Rueher M : Automatic Test Data Generation using Constraint Solving Techniques. Proc. ISSTA 98, ACM SIGSOFT, vol. 2, pp. 53-62, 1998.
10. Gotlieb A., Botella B. and Rueher M : A CLP Framework for Computing Structural Test Data Proc of Computational Logic (CL2000),pp. 399-413, 2000.
11. Keller C. W., Saha D., Basu S., Smolka S.A. : FocusCheck : A tool for Model Checking and Debugging Sequantial C Programs. TACAS 2005, LNCS 3440, pp 563-569, 2005
12. Kon O. and Castanet R. : Test generation for interworking systems. Computer Communications,vol. 23, pp. 642–652, 2000.
13. Krzystof R. Apt : Principles of Constraint Programming Cambridge University Press,2003.
14. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
15. Leavens Gary T. and Cheon Yoonsik : Design by Contract with JML. www.jmlspecs.org, August 2005.
16. A. Mackworth : Consistency in networks of relations. *Journal of Artificial Intelligence*, pages 8(1):99–118, 1977.
17. Malay K. Ganai, Aarti Gupta, Pranav Ashar: DiVer: SAT-Based Model Checking Platform for Verifying Large Scale Systems. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440,pp. 575–580, 2005.
18. Michela Milano (editor): Constraint and integer programming Kluwer Academic Publisher, 2004.
19. U. Montanari : Networks of constraints : Fundamental properties and applications to image processing. *Information science*, 7:95–132, 1974.
20. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik,M: Chaff: Engineering an Efficient SAT Solver. Proc of DAC, pp. 530–535, 2001
21. Rina Dechter: Constraint Processing. Morgan Kaufmann publisher,2003
22. Sy N.T. and Deville Y.: Automatic test data generation for programs with integer and float variables. Proc of. 16th IEEE International Conference on Automated Software Engineering(ASE01), 2001.
23. Sy N.T. and Deville Y.: Consistency Techniques for interprocedural Test Data Generation. Proc. of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE03), Helsinki, Finland, 2003,

III.6.2 CPBPV : A Constraint-Programming Framework for Bounded Program Verification

Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. A Constraint-Programming Framework for Bounded Program Verification. Proc. of CP2008 , LNCS 5202, pp. 327-341,2008, Springer-Verlag.

CPBPV: A Constraint-Programming Framework For Bounded Program Verification

Hélène Collavizza[1], Michel Rueher[1], Pascal Van Hentenryck[2]

[1] Université de Nice–Sophia Antipolis, France ({helen,rueher}@polytech.unice.fr)
[2] Brown University, Box 1910, Providence, RI 02912 (pvh@cs.brown.edu)

Abstract. This paper studies how to verify the conformity of a program with its specification and proposes a novel constraint-programming framework for bounded program verification (CPBPV). The CPBPV framework uses constraint stores to represent the specification and the program and explores execution paths nondeterministically. The input program is partially correct if each constraint store so produced implies the post-condition. CPBPV does not explore spurious execution paths as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. CPBPV uses the rich language of constraint programming to express the constraint store. Finally, CPBPV is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and less general. Experimental results often produce orders of magnitude improvements over earlier approaches, running times being often independent of the variable domains. Moreover, CPBPV was able to detect subtle errors in some programs while other frameworks based on model checking have failed.

1 Introduction

This paper is concerned with software correctness, a critical issue in software engineering. It proposes a novel constraint-programming framework for bounded program verification (CPBPV), i.e., when the program inputs (e.g., the array lengths and the variable values) are bounded. The goal is to verify the conformity of a program with its specification, that is to demonstrate that the specification is a consequence of the program. The key idea of CPBPV is to use constraint stores to represent the specification and the program, and to non-deterministically explore execution paths over these constraint stores. This non-deterministic constraint-based symbolic execution incrementally refines the constraint store, which initially consists of the precondition. Non-determinism occurs when executing conditional or iterative instructions and the non-deterministic execution refines the constraint store by adding constraints coming from conditions and from assignments. The input program is partially correct if each constraint store produced by the symbolic execution implies the post-condition. It is important to emphasize that CPBPV considers programs with complete specifications and that verifying the conformity between a program and its specification requires to check (explicitly or implicitly) all executables paths. This is not the case in

model-checking tools designed to detect violations of some specific property, e.g., safety or liveness properties.

The CPBPV framework has a number of fundamental benefits. First, contrary to earlier work using constraint programming or SMT [2, 11, 12], CPBPV does not use predicate abstraction or explore spurious execution paths, i.e., paths that do not correspond to actual executions over inputs satisfying the pre-condition. CPBPV incrementally prunes execution paths early by detecting that the constraint store is not consistent. Second, CPBPV uses the rich language of constraint programming to express the constraint store, including arbitrary logical and threshold combination of constraints, the *element* constraint, and global/combinatorial constraints that express complex relationships on a set of variables. Finally, CPBPV is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and less general.

The CPBPV framework was evaluated experimentally on a series of benchmarks from program verification. Experimental results of our (slow) prototype often produce orders of magnitude improvements over earlier approaches, and indicate that the running times are often independent of the variable domains. Moreover, CPBPV was able to found subtle errors in some programs that some other verification frameworks based on model-checking could not detect.

The rest of the paper is organized as follows. Section 2 illustrates how CPBPV handles constraints store on a motivating example. Section 3 formalizes the CPBPV framework for a small programming language and Section 4 discusses the implementation issues. Section 5 presents experimental results on a number of verification problems, comparing our approach with state of the art model-checking based verification frameworks. Section 6 discusses related work in test generation, bounded program verification and software model checking. Section 7 summarizes the contributions and presents future research directions.

2 The Constraint-Programming Framework at Work

This section illustrates the CPBPV verifier on a motivating example, the binary search program. CPBPV uses Java programs and JML specifications for the pre- and post-conditions, appropriately enhanced to support the expressivity of constraint programming. Figure 1 depicts a binary search program to determine if a value v is present in a sorted array t. (Note that `\result` in JML corresponds to the value returned by the program). To verify this program, our prototype implementation requires a bound on the length of array t, on its elements, and on v. We will verify its correctness for specific lengths and simply assume that the values are signed integers on a number of bits.

The initial constraint store of the CPBPV verifier, assuming an input array of length 8, is the precondition[3] $c_{pre} \equiv \forall 0 \leq i < 7 : t^0[i] \leq t^0[i+1]$ where t^0 is an array of constraint variables capturing the input. The constraint variables are annotated with a version number as CPBPV performs a SSA-like renaming

[3] We omit the domain constraints on the variables for simplicity.

```
/*@ requires (\forall int i; i>=0 && i<t.length-1;t[i]<=t[i+1])
  @ ensures
  @   (\result != -1 ==> t[\result] == v) &&
  @   (\result == -1 ==> \forall int k; 0 <= k < t.length ; t[k] != v) @*/
1 static int binary_search(int[] t, int v) {
2    int l = 0;
3    int u = t.length-1;
4    while (l <= u) {
5      int m = (l + u) / 2;
6      if (t[m]==v)
7          return m;
8      if (t[m] > v)
9         u = m - 1;
10      else
11        l = m + 1;    }   // ERROR else u = m - 1;
12   return -1; }
```

Fig. 1. The Binary Search Program

[10] on the fly since each assignment generates constraints possibly linking the old and the new values of the assigned variable. The assignments in lines 2–3 add the constraints $l^0 = 0 \wedge u^0 = 7$. CPBPV then considers the loop instruction. Since $l^0 \leq u^0$, it enters the loop body, adds the constraint $m^0 = (l^0 + u^0)/2$, which simplifies to $m^0 = 3$, and considers the conditional statement on line 6. The execution of the statement is nondeterministic: Indeed, both $t^0[3] = v^0$ and $t^0[3] \neq v^0$ are consistent with the constraint store, so that the two alternatives, which give rise to two execution paths, must be explored. Note that these two alternatives correspond to actual execution paths in which $t[3]$ in the input is equal to, or different from, input v. The first alternative adds the constraint $t^0[3] = v^0$ to the store and executes line 7 which adds the constraint $result = m^0$. CPBPV has thus obtained an execution path p whose final constraint store c_p is: $c_{pre} \wedge l^0 = 0 \wedge u^0 = 7 \wedge m^0 = (l^0 + u^0)/2 \wedge t^0[m^0] = v^0 \wedge result = m^0$
CPBPV then checks whether this store c_p implies the post-condition c_{post} by searching for a solution to $c_p \wedge \neg c_{post}$. This test fails, indicating that the computation path p, which captures the set of actual executions in which $t[3] = v$, satisfies the specification. CPBPV then explores the other alternatives to the conditional statement in line 6. It adds the constraint $t^0[m^0] \neq v^0$ and executes the conditional statement in line 8. Once again, this statement is nondeterministic. Its first alternative assumes that the test holds, generating the constraint $t^0[m^0] > v^0$ and executing the instruction in line 9. Since u is (re-)assigned, CPBPV creates a new variable u^1 and posts the constraint $u^1 = m^0 - 1 = 2$. The execution returns to line 4, where the test now reads $l^0 \leq u^1$, since CPBPV always uses the most recent version for each variable. Since the constraint stores entails $l^0 \leq u^1$, the only extension to the current path consists of executing line 5, adding the constraint $m^1 = (l^0 + u^1)/2$, which actually simplifies to $m^1 = 1$. Another complete execution path is then obtained by executing lines 6 and 7.

Consider now a version of the program in which line 11 is replaced by `u = m-1`. To illustrate the CPBPV verifier, we specify partial execution paths by indicating which alternative is selected for each nondeterministic instruction. For instance, $\langle T_4, F_6, T_8, T_5, T_6\rangle$ denotes the last execution path discussed above in which the true alternative is selected for the first execution of the instruction in line 4, the false alternative for the first execution of instruction 6, the true alternative for the first instruction of instruction 8, the true alternative of the second execution of instruction 5, and the true alternative of the second execution of instruction 6. Consider the partial path $\langle T_4, F_6, F_8\rangle$ and let us study how it can be extended. The partial path $\langle T_4, F_6, F_8, T_4, T_6\rangle$ is not explored, since it produces a constraint store containing

$$c_{pre} \ \wedge \ t^0[3] \neq v^0 \ \wedge \ t^0[3] \leq v^0 \ \wedge \ t^0[1] = v^0$$

which is clearly inconsistent. Similarly, the path $\langle T_4, F_6, F_8, T_4, F_6, T_8\rangle$ cannot be extended. The output of CPBPV on this incorrect program when executed on an array of length 8 (with integers coded on 8-bits to make it readable) produces, in 0.025 seconds, the counterexample:
$v^0 = -126 \ \wedge \ t^0 = [-128, -127, -126, -125, -124, -123, -122, -121] \ \wedge \ result = -1.$

This example highlights a few interesting benefits of CPBPV.

1. The verifier only considers paths that correspond to collections of actual inputs (abstracted by constraint stores). The resulting execution paths must all be explored since our goal is to prove the partial correctness of the program.
2. The performance of the verifier is independent of the integer representation on this application: it only requires a bound on the length of the array.
3. The verifier returns a counter-example for debugging the program.

Note that $CBMC$ and $ESC/Java2$, two state-of-the-art model checkers fail to verify this example as discussed in Section 5.

3 Formalization of the Framework

This section formalizes the CPBPV verifier on a small abstract language using a small-step SOS semantics. The semantics primarily specifies the execution paths over constraint stores explored by the verifier. It features `assert` and `enforce` constructs which are necessary for modular composition.

Syntax Figure 2 depicts the syntax of the programs and the constraints generated by the verifier. In the following, we use s, possibly subscripted, to denote elements of a syntactic entity S.

Renamings CPBPV creates variables and arrays of variables "on-the-fly" when they are needed. This process resembles an SSA normalization but does not introduce the join nodes, since the results of different execution paths are not merged. Similar renamings are used in model checking. The renaming uses mappings of type $V \cup A \rightarrow \mathcal{N}$ which maps variables and arrays into a natural numbers

L : *list of instructions*; I : *instructions*; B : *Boolean expressions*
E : *integer expressions*; A : *arrays*; V : *variables*

$L ::= I; L \mid \epsilon$
$I ::= A[E] \leftarrow E \mid V \leftarrow E \mid \mathit{if}\ B\ I \mid \mathit{while}\ B\ I \mid \mathit{assert}(B) \mid \mathit{enforce}(B) \mid \mathit{return}\ E \mid \{L\}$
$B ::= \mathit{true} \mid \mathit{false} \mid E > E \mid E \geq E \mid E = E \mid E \neq E \mid E \leq E \mid E < E$
$B ::= \neg B \mid B \wedge B \mid B \vee B \mid B \Rightarrow B$
$E ::= V \mid A[E] \mid E + E \mid E - E \mid E \times E \mid E/E \mid$

C : *constraints* $\qquad E^+$: *solver expressions*
$V^+ = \{v^i \mid v \in V\ \&\ i \in \mathcal{N}\}$: *solver variables*
$A^+ = \{a^i \mid a \in A\ \&\ i \in \mathcal{N}\}$: *solver arrays*

$C ::= \mathit{true} \mid \mathit{false} \mid E^+ > E^+ \mid E^+ \geq E^+ \mid E^+ = E^+ \mid E^+ \neq E^+ \mid E^+ \leq E^+ \mid E^+ < E^+$
$C ::= \neg C \mid C \wedge C \mid C \vee C \mid C \Rightarrow C$
$E^+ ::= V \mid A[E^+] \mid E^+ + E^+ \mid E^+ - E^+ \mid E^+ \times E^+ \mid E^+/E^+ \mid$

Fig. 2. The Syntax of Programs and Constraints

denoting their current "version numbers". In the semantics, the version number is incremented each time a variable or an array element is assigned. We use $\sigma_\perp$ to denote the uniform mapping to zero (i.e., $\forall x \in V \cup A : \sigma_\perp(x) = 0$) and $\sigma[x/i]$ the mapping σ where x now maps to i, i.e., $\sigma[x/i](y) = \mathit{if}\, x = y\ \mathit{then}\ i\ \mathit{else}\ \sigma(y)$. These mappings are used by a polymorphic renaming function ρ to transform program expressions into constraints. For example, $\rho\ \sigma\ b_1 \oplus b_2 = (\rho\ \sigma\ b_1) \oplus (\rho\ \sigma\ b_2)$(where $\oplus \in \{\wedge, \vee, \Rightarrow\}$) is the rule used to transform a logical expression.

Configurations The CPBCV semantics mostly uses configurations of the type $\langle l, \sigma, c\rangle$, where l is the list of instructions to execute, σ is a version mapping, and c is the set of constraints generated so far. It also uses configurations of the form $\langle \top, \sigma, c\rangle$ to denote final states and configurations of the form $\langle \perp, \sigma, c\rangle$ to denote the violation of an assertion. The semantics is specified by rules of the form $\frac{\text{conditions}}{\gamma_1 \longmapsto \gamma_2}$ stating that configuration γ_1 can be rewritten into γ_2 when the conditions hold.

Conditional Instructions The conditional instruction *if b i* considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then the store is augmented with c_b and the body is executed. If the negation $\neg c_b$ is consistent with the store, then the constraint store is augmented with $\neg c_b$. Both rules may apply, since the store may represent some memory states satisfying the condition and some violating it.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle} \qquad \frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

Iterative Instructions The while instruction *while b i* also considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then

the constraint store is augmented with c_b, the body is executed, and the while instruction is reconsidered. If the negation $\neg c_b$ is consistent with the constraint store, then the constraint store is augmented with $\neg c_b$.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i; \mathit{while}\ b\ i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

$$\frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

Scalar Assignments Scalar assignments create a new constraint variable for the program variable to be assigned and add a constraint specifying that the variable is equal to the right-hand side. A new renaming mapping is produced.

$$\frac{\sigma_2 = \sigma_1[v/\sigma_1(v)+1]\ \ \&\ \ c_2 \equiv (\rho\ \sigma_2\ v) = (\rho\ \sigma_1\ e)}{\langle v \leftarrow e\ ;\ l, \sigma_1, c_1\rangle \longmapsto \langle l, \sigma_2, c_1 \wedge c_2\rangle}$$

Assignments of Array Elements The assignment of an array element creates a new constraint array, add a constraint for the index being indexed and posts constraints specifying that all the new constraint variables in the array are equal to their earlier version, except for the element being indexed. Note that the index is an expression which may contain variables as well, giving rise to the well-known *element* constraint in constraint programming [25].

$$\frac{\begin{array}{l}\sigma_2 = \sigma_1[a/\sigma_1(a)+1] \\ c_2 \equiv (\rho\ \sigma_2\ a)[\rho\ \sigma_1\ e_1] = (\rho\ \sigma_1\ e_2) \\ c_3 \equiv \forall i \in 0..a.\mathit{length} : (\rho\ \sigma_1\ e_1) \neq i\ \Rightarrow\ (\rho\ \sigma_2\ a)[i] = (\rho\ \sigma_1\ a)[i]\end{array}}{\langle a[e_1] \leftarrow e_2, \sigma_1\ ;\ l, c_1\rangle \longmapsto \langle l, \sigma_2, c_1 \wedge c_2 \wedge c_3\rangle}$$

Assert Statements An assert statement checks whether the assertion is implied by the control store in which case it proceeds normally. Otherwise, it terminates the execution with an error.

$$\frac{c \Rightarrow (\rho\ \sigma\ b)}{\langle \mathit{assert}\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c\rangle} \qquad \frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{assert}\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle \bot, \sigma, c\rangle}$$

Enforce Statements An enforce statement adds a constraint to the constraint store if it is satisfiable.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{enforce}\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

Block Statements Block statements simply remove the braces.

$$\langle \{l_1\}\ ;\ l_2, \sigma, c\rangle \longmapsto \langle l_1 : l_2, \sigma, c\rangle$$

Return Statements A return statement simply constrains the *result* variable.

$$\frac{c_2 \equiv (\rho\ \sigma_1\ result) = (\rho\ \sigma_1\ e)}{\langle return\ e\ ;\ l, \sigma_1, c_1\rangle \longmapsto \langle \sigma_1, c_1 \wedge c_2\rangle}$$

Termination Termination also occurs when no instruction remains.

$$\langle \epsilon, \sigma, c\rangle \longmapsto \langle \top, \sigma, c\rangle$$

The CPBPV Semantics Let $\mathcal{P}$ be program $b_{pre}\ l\ b_{post}$ in which b_{pre} denotes the precondition, l is a list of instructions, and b_{post} the post-condition. Let $\stackrel{*}{\longmapsto}$ be the transitive closure of $\longmapsto$. The final states are specified by the set

$$SFN(b_{pre}, \mathcal{P}) = \{\ \langle f, \sigma, c\rangle | \langle i, \sigma_\perp, \rho\ \sigma_\perp\ b_{pre}\rangle \stackrel{*}{\longmapsto} *\langle f, \sigma, c\rangle\ \wedge\ f \in \{\perp, \top\}\ \}$$

The program violates an assertion if the set

$$SFE(b_{pre}, \mathcal{P}, b_{post}) = \{\langle \perp, \sigma, c\rangle \in SFN(b_{pre}, \mathcal{P})\}$$

is not empty. It violates its specification if the set

$$SFE(b_{pre}, \mathcal{P}, b_{post}) = \{\top, \sigma, c\rangle \in SFN(b_{pre}, \mathcal{P}) \mid c\ \wedge\ (\rho\ \sigma\ \neg b_{post})\ \text{satisfiable}\}$$

is not empty. It is partially correct otherwise.

4 Implementation issues

The CPBPV framework is parametrized by a list of solvers $(S_1, \ldots, S_k)$ which are tried in sequence, starting with the least expensive and less general. When checking satisfiability, the verifier never tries solver $S_{i+1}, \ldots, S_k$ if solver S_i is a decision procedure for the constraint store. If solver S_i is not a decision procedure, it uses an abstraction α of the constraint store c satisfying $c \Rightarrow \alpha$ and can still detect failed execution paths quickly. The last solver in the sequence is a constraint-programming solver (CP solver) over finite domains which iterates pruning and searching to find solutions or prove infeasibility. When the CP solver makes a choice, the earlier solvers in the sequence are called once again to prune the search space or find solutions if they have become decision procedures. Our prototype implementation uses a sequence (MIP, CP), where MIP is the mixed integer-programming tool ILOG CPLEX[4] and CP is the constraint-programming tool Ilog JSOLVER. Our Java implementation also performs some trivial simplifications such as constant propagation but is otherwise not optimized in its use of the solvers and in its renaming process whose speed and memory usage could be improved substantially. Practically, simplifications are done on the fly and the MIP solver is called at each node of the executable paths. The CP solver is only called at the end of the executable paths when

[4] See http://www.ilog.com/products.

the complete post condition is considered. Currently, the implementation use a depth-first strategy for the CP solver, but modern CP languages now offer high-level abstractions to implement other exploration strategies. In practice, when CPBPV is used for model checking as discussed below, it is probably advisable to use a depth-first iterative deepening implementation.

5 Experimental results

In this section, we report experimental results for a set of traditional benchmarks for program verification. We compare CPBVP with the following frameworks:

- ESC/Java is an Extended Static Checker for Java to find common run-time errors in JML-annotated Java programs by static analysis of the code and its annotations. See http://kind.ucd.ie/products/opensource/ESCJava2/.
- CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It allows for the verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. See http://www.cprover.org/cbmc/.
- BLAST, the Berkeley Lazy Abstraction Software Verification Tool, is a software model checker for C programs. See http://mtc.epfl.ch/software-tools/blast/.
- EUREKA is a C bounded model checker which uses an SMT solver instead of an SAT solver. See http://www.ai-lab.it/eureka/.
- Why is a software verification platform which integrates many existing provers (proof assistants such as Coq, PVS, HOL 4,...) and decision procedures such as Simplify, Yices, ...). See http://why.lri.fr/.

Of course, neither the expressiveness nor the objectives of all these systems are the same as the one of CPBPV. For instance, some of them can handle CTL/LTL constraints whereas CPBPV dos not yet support this kind of constraints. Nevertheless, this comparison is useful to illustrate the capabilities of CPBPV.
All experiments were performed on the same machine, an Intel(R) Pentium(R) M processor 1.86GHz with 1.5G of memory, using the version of the verifiers that can be downloaded from their web sites (except for EUREKA for which the execution times given in [2, 3] are reported.) For each benchmark program, we describe the data entries and the verification parameters. In the tables, "UNABLE" means that the corresponding framework is unable to validate the program either because a lack of expressiveness or because of time or memory limitations, "NOT_FOUND" that it does not detect an error, and "FALSE_ERROR" that it reports an error in a correct program. Complete details of the experiments, including input files and error traces, can be found in [13].

Binary search We start with the binary search program presented in figure 1. ESC/Java is applied on the program described in Figure 1. ESC/Java requires a limit on the number of loop unfoldings, which we set to $log(n) + 1$ which is the worst case complexity of binary search algorithm for an array of length n. Similarly, CBMC requires an overestimate of the number of loop unfoldings. Since

CPBPV	array length	8	16	32	64	128	256
	time	1.081s	1.69s	4.043s	17.009s	136.80s	1731.696s
CBMC	array length	8	16	32	64	128	256
	time	1.37s	1.43s	UNABLE	UNABLE	UNABLE	UNABLE
Why	with invariant	11.18s					
	without invariant	UNABLE					
ESC/Java	FALSE_ERROR						
BLAST	UNABLE						

Table 1. Comparison table for binary search

CBMC does not support first-order expressions such as JML $\backslash forall$ statement, we generated a C program for each instance of the problem (i.e., each array length). For example, the postcondition for an array of length 8 is given by

```
(result!=-1 && a[result]==x)||
(result==-1 && (a[0]!=x&&a[1]!=x&&a[2]!=x&&a[3]!=x&&a[4]!=x&&a[5]!=x&&a[6]!=x&&a[7]!=x)
```

For the Why framework, we used the binary search version given in their distribution. This program uses an assert statement to give a loop invariant.

Note that CPBPV does not require any additional information: no invariant and no limits on loop unfoldings. During execution, it selects a path by nondeterministically applying the semantic rules for conditional and loop expressions.

Table 1 reports the experimental results. Execution times for CPBPV are reported as a function of the array length for integers coded on 31 bits.[5] Our implementation is neither optimized for time or space at this stage and times are only given to demonstrate the feasibility of the CPBPV verifier.

The "Why" framework [16] was unable to verify the correctness without the loop invariant; 60% of the proof obligations remained unknown.

The CBMC framework was not able to do the verification for an instance of length 32 (it was interrupted after 6691,87s).

ESC/Java was unable to verify the correctness of this program unless complete loop invariants are provided [6].

An Incorrect Binary search Table 2 reports experimental results for an incorrect *binary search* program (see Figure 1, line 11) for CPBPV, ESC/Java, CBMC, and Why using an invariant. The error trace found with CPBPV has been described in Section 2. The error traces provided by CBMC and ESC/Java only show the decisions taken along the faulty path can be found in [13]. In contrast to CPBPV, they do not provide any value for the array nor the searched data. Observe that CPBPV provides orders of magnitude improvements in efficiency over CBMC and also outperforms ESC/Java by almost a factor 8 on the largest instance.

[5] The commercial MIP solver fails with 32-bit domains because of scaling issues.

[6] a version with loop invariants that allows to show the correctness of this program has been written by David Cok, a developper of ESC/Java, after we contacted him.

	CPBPV	ESC/Java	CBMC	WHY with invariant	BLAST
length 8	0.027s	1.21 s	1.38s	NOT_FOUND	UNABLE
length 16	0.037s	1.347 s	1.69s	NOT_FOUND	UNABLE
length 32	0.064s	1.792 s	7.62s	NOT_FOUND	UNABLE
length 64	0.115s	1.886 s	27.05s	NOT_FOUND	UNABLE
length 128	0.241s	1.964 s	189.20s	NOT_FOUND	UNABLE

Table 2. Experimental Results for an Incorrect Binary Search

	CPBPV	ESC/Java	CBMC	Why	BLAST
time	0.287s	1.828s	0.82s	8.85s	UNABLE

Table 3. Experimental Results on the Tritype Program

The Tritype Program The tritype program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths: only 10 paths correspond to actual inputs because of complex conditional statements in the program. The program takes three positive integers as inputs (the triangle sides) and returns 2 if the inputs correspond to an isosceles triangle, 3 if they correspond to an equilateral triangle, 1 if they correspond to some other triangle, and 4 otherwise. The `tritype` program in Java with its specification in JML can be found in[13]. Table 3 depicts the experimental results for CPBPV, ESC/Java, CBMC, BLAST and Why. BLAST was unable to validate this example because the current version does not handle linear arithmetic. Observe the excellent performance of CPBPV and note that our previous approach using constraint programming and Boolean abstraction to abstract the conditions, validated this benchmark in 8.52 seconds when integers were coded on 16 bits [12]. It also explored 92 spurious paths.

An Incorrect Tritype Program Consider now an incorrect version of *Tritype* program in which the test *"if ((trityp==2)&&(i+k>j))"* in line 22 (see [13]) is replaced by *"if ((trityp==1)&&(i+k>j))"*. Since the local variable *trityp* is equal to *2* when *i==k*, the condition *(i+k)>j* implies that *(i,j,k)* are the sides of an isosceles triangle (the two other triangular inequalities are trivial because j>0). But, when *trityp=1*, *i==j* holds and this incorrect version may answer that the triangle is isosceles while it may not be a triangle at all. For example, it will return *2* when *(i,j,k)=(1,1,2)*. Table 4 depicts the experimental results. Execution times correspond to the time required to find the first error. The error found with CPBPV corresponds to input values $(i, j, k) = (1, 1, 2)$ mentioned earlier. Once again, observe the excellent behavior of CPBPV compared to the remaining tools. [7]

[7] For CBMC, we have contacted D. Kroening who has recommended to use the option CPROVER_assert. If we do so, CBMC is able to find the error, but we must add

	CPBPV	ESC/Java	CBMC	WHY
time	0.056s s	1.853s	NOT_FOUND	NOT_FOUND

Table 4. Experimental Results for the Incorrect Tritype Program

	CPBPV	ESC/Java	CBMC	EUREKA
length 8	1.45s	3.778 s	1.11s	91s
length 16	2.97s	UNABLE	2.01s	UNABLE
length 32	UNABLE	UNABLE	6.10s	UNABLE
length 64	UNABLE	UNABLE	37.65s	UNABLE

Table 5. Experimental Results for Bubble Sort

Bubble Sort with initial condition This benchmark (see [13]) is taken from [2] and performs a bubble sort of an array t which contains integers from 0 to $t.length$ given in decreasing order. Table 5 shows the comparative results for this benchmark. CPBPV was limited on this benchmark because its recursive implementation uses up all the JAVA stack space. This problem should be remedied by removing recursion in CPBPV.

Selection Sort We now present a benchmark to highlight both modular verification and the `element` constraint of constraint programming to index arrays with arbitrary expressions. The benchmark described in [13]. Assume that function `findMin` has been verified for arbitrary integers. When encountering a call to `findMin`, CPBPV first checks if its precondition is entailed by the constraint store, which requires a consistency check of the constraint store with respect to the negation of the precondition. Then CPBPV replaces the call by the post-condition where the formal parameters are replaced by the actual variables. In particular, for the first iteration of the loop and an array length of 40, CPBPV generates the conjunction $0 \leq k^0 < 40 \ \wedge \ t^0[k^0] \leq t^0[0] \ \wedge \ \ldots \ \wedge \ t^0[k^0] \leq t^0[39]$ which features `element` constraint [25]. Indeed, k^0 is a variable and a constraint like $t^0[k^0] \leq t^0[0]$ indexes the array t^0 of variables using k^0.

The modular verification of the selection sort explores only a single path, is independent of the integer representation, and takes less than $0.01s$ for arrays of size 40. The bottleneck in verifying selection sort is the validation of function `findMin`, which requires the exploration of many paths. However the complete validation of selection sort takes less than 4 seconds for an array of length 6. Once again, this should be contrasted with the model-checking approach of Eureka [2]. On a version of selection sort where all variables are assigned specific values (contrary to our verification which makes no assumptions on the inputs), Eureka takes 104 seconds on a faster machine. Reference [2] also reports that CBMC

some assumptions to mean that there is no overflow into the sums, in order to prove the correct version of tritype with this same option.

takes 432.6 seconds, that BLAST cannot solve this problem, and that SATABS [9] only verifies the program for an array with 2 elements.

Sum of Squares Our last benchmark is described in [13] and computes the sum of the square of the n first integers stored in an array. The precondition states that n is the size of the array and that t must contain any possible permutation of the n first integers. The postcondition states that the result is $n \times (n + 1) \times (2 \times n + 1)/6$. The benchmark illustrates two functionalities of constraint programming: the ability of specifying combinatorial constraints and of solving nonlinear problems. The `alldifferent` constraint[23] in the precondition specifies that all the elements of the array are different, while the program constraints and postcondition involves quadratic and cubic constraints. The maximum instance that we were able to solve with CPBPV was an array of size 10 in 66.179s.

CPLEX, the MIP solver, plays a key role in all these benchmarks. For instance, the CP solver is never called in the Tritype benchmark. For the Binary search benchmark, there are length calls to the CP solver but almost 75% of the CPU time is spent in the CP solver. Since there is only path in the Buble sort benchmark, the CP solver is only called once. In the Sum of squares example, 80% of the CPU time is spent in the CP solver.

6 Discussion and Related Work

We briefly review recent work in constraint programming and model checking for software testing, validation, and verification. We outline the main differences between our CPBPV framework and existing approaches.

Constraint Logic Programming Constraint logic programming (CLP) was used for test generation of programs (e.g., [17, 20, 24, 19]) and provides a nice implementation tool extending symbolic execution techniques [4]. Gotlieb et al. showed how to represent imperative programs as constraint logic programs and used predicate abstraction (from model checking) and conditional constraints within a CLP framework. Flanagan [15] formalized the translation of imperative programs into CLP, argued that it could be used for bounded model checking, but did not provide an implementation. The test-generation methodology was generalized and applied to bounded program verification in [11, 12]. The implementation used dedicated predicate abstractions to reduce the exploration of spurious execution paths. However, as shown in the paper, the CPBPV verifier is significantly more efficient and often avoids the generation of spurious execution paths completely.

Model Checking It is also useful to contrast the CPBPV verifier with model-checking of software systems. SAT-based bounded model checking for software[6] consists in building a propositional formula whose models correspond to execution paths of bounded length violating some properties and in using SAT

solvers to check whether the resulting formula is satisfiable. SAT-based model-checking platforms [6] have been widely popular thanks to significant progress in SAT solvers. A fundamental issue faced by model checkers is the state space explosion of the resulting model. Various techniques have been proposed to address this challenge, including generalized symbolic execution (e.g., [21]), SMT-based model checking, and abstraction/refinement techniques. SMT-based model checking is the idea of representing and checking quantifier-free formulas in a more general decidable theory (e.g. [18, 14, 22]). These SMT solvers integrate dedicated solvers and share some of the motivations of constraint programming. Predicate abstraction is another popular technique to address the state space explosion. The idea consists in abstracting the program to obtain an abstract program on which model checking is performed. The model checker may then generate an abstract counterexample which must be checked to determine if it corresponds to a concrete execution path. If the counterexample is spurious, the abstract program is refined and the process is iterated. A successful predicate abstraction consists of abstracting the concrete program into a Boolean program (e.g., [5, 7, 8]). In recent work [3, 2], Armando & al proposed to abstract concrete programs into linear programs and used an abstraction of sets of variables and array indices. They showed that their tool compares favourably and, on some of the programs considered in this paper, outperforms model checkers based on predicate abstraction.

Our CPBPV verifier contrasts with SAT-based model checkers, SMT-based model checkers and predicate abstraction based approaches: It does not abstract the program and does not generate spurious execution paths. Instead it uses a constraint-solver and nondeterministic exploration to incrementally construct abstractions of execution paths. The abstraction uses constraint stores to represent sets of concrete stores. On many bounded verification benchmarks, our preliminary experimental results show significant improvements over the state-of-the-art results in [2]. Model checking is well adapted to check low-level C program and hardware applications with numerous Boolean constraints and bit-wise operations: It was successfully used to compare an ANSI C program with a circuit given as design in Verilog [7]. However, it is important to observe that in model checking, one is typically interested in checking some specific properties such as buffer overflows, pointer safety, or user-specified assertions. These properties are typically much less detailed than our post-conditions and abstracting the program may speed up the process significantly. In our CPBPV verifier, it is critical to explore all execution paths and the main issue is how to effectively abstract memory stores by constraints and how to check satisfiability incrementally. It is an intriguing issue to determine whether an hybridization of the two approaches would be beneficial for model checking, an issue briefly discussed in the next section. Observe also that this research provides convincing evidence of the benefits of Nieuwenhuis' challenge [22] aiming at extending SMT[8] with CP techniques.

[8] See also [1] for a study of the relations between constraint programming and Satisfiability Modulo Theories (SMT)

7 Perspectives and Future Work

This paper introduced the CPBPV framework for bounded program verification. Its novelty is to use constraints to represent sets of memory stores and to explore execution paths over these constraint stores nondeterministically and incrementally. The CPBPV verifier exploits the fact that, when variables and arrays are bounded, the constraint store can always be checked for feasibility. As a result, it never explores spurious execution path contrary to earlier approaches combining constraint programming and predicate abstraction [11, 12] or integrating SMT solvers and the abstraction/refinement approach from model checking [2]. We demonstrated the CPBPV verifier on a number of standard benchmarks from model checking and program checking as well as on nonlinear programs and functions using complex array indexings, and showed how to perform modular verification. The experimental results demonstrate the potential of the approach: The CPBPV verifier provides significant gain in performance and functionalities compared to other tools.

Our current work aims at improving and generalizing the framework and implementation. In particular, we would like to include tailored, light-weight solvers for a variety of constraint classes, the optimization of the array implementation, and the integration of Java objects and references. There are also many research avenues opened by this research, two of which are reviewed now.

Currently, the CPBPV verifier does not check for variable overflows: the constraint store enforces that variables take values inside their domains and execution paths violating these constraints are thus not considered. It is possible to generalize the CPBPV verifier to check overflows as the verification proceeds. The key idea is to check before each assignment if the constraint store entails that the value produced fits in the selected integer representation and generate an error otherwise. (Similar assertions must in fact be checked for each subexpression in the right hand-side in the language evaluation order. Interval techniques on floats [4] may be used to obtain conservative checking of such assertions.

An intriguing direction is to use the CPBPV approach for properties checking. Given an assertion to be verified, one may perform a backward execution from the assertion to the function entry point. The negation of the assertion is now the pre-condition and the pre-condition becomes the post-condition. This requires to specify inverse renaming and executions of conditional and iterative statements but these have already been studied in the context of test generation.

Acknowledgements Many thanks to Jean-Franois Couchot for many helps on the use of the *Why* framework.

References

1. Aït-Kaci H., Berstel B., Junker U., Leconte M., Podelski A. : Satisfiability Modulo Structures as Constraint Satisfaction : An Introduction. Procs of JFLA 2007.
2. Armando A., Benerecetti M., and Montovani J. Abstraction Refinement of Linear Programs with Arrays. Proceedings of TACAS 2007, LNCS 4424: 373–388.

3. Armando A., Mantovani J., and Platania L. Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver. Proc. SPIN'06. LNCS 3925, Pages 146-162.
4. Botella B., Gotlieb A., Michel C. Symbolic execution of floating-point computations. Software Testing, Verification and Reliability. 16:2:97–121.2006.
5. Thomas Ball, Andreas Podelski, Sriam K. Rajamani Boolean and Cartesian Abstraction for Model Checking C Programs. Proc. of TACAS 2001.
6. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking using Satisfiability Solving. FMSD, 19(1):7–34, 2001.
7. Clarke E., Kroening D., Lerda F. : A Tool for Checking ANSI-C programs. Tacas 2004, LNCS 2988, pp 168-176, 2004
8. Clarke E., Kroening D., Sharygina N., Yorav K. : Predicate abstraction of ANSI-C Programs using SAT. FMSD, 25:105–127, 2004
9. Clarke E., Kroening D., Sharygina N., Yorav K. : SATABS: SAT-Based Predicate Abstraction for ANSI-C. TACAS'05, 570–574, 2005.
10. Cytron R., Ferrante J., Rosen B., Wegman M., and Zadeck K. : Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
11. Collavizza H. and Rueher M. : Software Verification using Constraint Programming Techniques. Procs of TACAS 2006, LNCS 3920: 182-196.
12. Collavizza H. and Rueher M. : Exploring different constraint-based modelings for program verification Procs of CP 2007, LNCS 3920: 182-196
13. Collavizza H. Rueher M., Van Hentenryck P. : Comparison between CPBPV with ESC/Java, CBMC, Blast, EUREKA and Why. http://www.i3s.unice.fr/~rueher/verificationBench.pdf
14. Bruno Dutertre and Leonardo Mendonca de Moura. A fast linear-arithmetic solver for DPLL(T). CAV 2006, pages 81–94. LNCS 4144.
15. Cormac Flanagan, "Automatic software model checking via constraint logic" (2004). Science of Computer Programming. 50 (1-3), pp. 253-270.
16. Fillitre J.C., Claude March.The Why/Krakatoa/Caduceus Platform for Deductive Program Verification Proc. CAV'2007, LNCS 4590. pp 173-177.
17. Gotlieb A., Botella B. and Rueher M : Automatic Test Data Generation using Constraint Solving Techniques. Proc. ISSTA 98, ACM SIGSOFT (2), 1998.
18. Ganzinger H., Hagen G., Nieuwenhuis R.,Oliveras A., and Tinelli C.: DPLL(T): Fast Decision Procedures. Proc. of CAV 2004, 175-188, 2004.
19. P. Godefroid, M. Y. Levin, D. Molnar: Automated Whitebox Fuzz Testing, NDSS 2008, Network and Distributed System Security Symposium.
20. Daniel Jackson and Mandana Vaziri, Finding Bugs with a Constraint Solver, ACM SIGSOFT Symposium on Software Testing and Analysis, 14–15, 2000.
21. Khurshid, S., Pasareanu, C.S., and Vissser, W. "Generalized Symbolic Execution for Model Checking and Testing", in TACAS 2003, Warsaw, Poland.
22. R. Nieuwenhuis, A. Oliveras, E. Rodrguez-Carbonell and A. Rubio: Challenges in Satisfiability Modulo Theories. Invited Talk. RTA 2007, LNCS 4533, pp 2-18.
23. J-C. Régin. A filtering algorithm for constraints of difference in CSPs. AAAI-94, Seattle, WA, USA, pp 362–367, 1994.
24. Sy N.T. and Deville Y.: Automatic Test Data Generation for Programs with Integer and Float Variables. Proc of. 16th IEEE ASE01, 2001.
25. VanHentenryck P. (1989) Constraint Satisfaction in Logic Programming, MIT Press.
26. Numerica: A Modeling Language for Global Optimization Pascal Van Hentenryck, Laurent Michel, Yves Deville. MIT Press, 1997.

Bibliographie

[1] J. R. Abrial : *The B-Book. Assigning Programs to Meanings.* Cambridge University Press, 1986.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager et P. H. Schmitt : The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] Krzystof R. Apt : *Principles of Constraint Programming.* Cambridge University Press, 2003.

[4] A. Armando, M. Benerecetti et J. Mantovani : Abstraction refinement of linear programs with arrays. *In TACAS*, volume 4424 de *LNCS*, pages 373–388. Springer-Verlag, 2007.

[5] A. Armando, J. Mantovani et L. Platania : Bounded model checking of C programs using a SMT solver instead of a SAT solver. *In SPIN workshop*, volume 3925 de *LNCS*, pages 146–162. Springer-Verlag, 2006.

[6] A. Armando, J. Mantovani et L. Platania : Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, février 2009.

[7] D. Beyer, T.A. Henzinger, R. Jhala et R. Majumdar : The software model checker BLAST : Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9:505–525, 2007.

[8] B. Botella, A. Gotlieb et C. Michel : Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[9] B. Botella, A. Gotlieb, C. Michel, M. Rueher et P. Taillibert : Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et sciences informatiques*, 21:1163–1187, 2002.

[10] F. Bouquet, F. Dadeau, B. Legeard et M. Utting : JML-Testing-Tools : a symbolic animator for JML specifications using CLP. *In 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 de *LNCS*, pages 551–556. Springer-Verlag, 2005.

[11] L. Burdy, A. Requet et J. Lanet : Java applet correctness : a developer-oriented approach. *In Formal Methods (FME'03)*, volume 2805 de *LNCS*, pages 422–439. Springer Verlag, 2003.

[12] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino et Erik Poll : An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[13] E. Clarke, D. Kroening et F. Lerda : A tool for checking ansi-c programs. *In TACAS 2004*, volume 2988 de *LNCS*, pages 168–176. Springer-Verlag, 2004.

[14] E. Clarke, D. Kroening, N. Sharygina et K. Yorav : Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25:105–127, 2004.

[15] E. Clarke, D. Kroening, N. Sharygina et K. Yorav : SATABS : SAT-based predicate abstraction for ANSI-C. *In TACAS, Tool session*, volume 3440 de *LNCS*, pages 570–574. Springer-Verlag, 2005.

[16] David R. Cok et Joseph Kiniry : ESC/Java2 : Uniting ESC/Java and JML. *In* Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet et Traian Muntean, éditeurs : *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS*, volume 3362 de *Lecture Notes in Computer Science*, pages 108–128. Springer, 2005.

[17] H. Collavizza et M. Rueher : Software verification using constraint programming techniques. *In TACAS*, volume 3920 de *LNCS*, pages 182–196. Springer-Verlag, 2006.

[18] H. Collavizza et M. Rueher : Exploring different constraint-based modelings for program verification. *In Constraint Programming International Conference*, volume 4741 de *LNCS*, pages 49–63. Springer-Verlag, 2007.

[19] H. Collavizza, M. Rueher et P. Van Hentenryck : A constraint-programming framework for bounded program verification. *In Constraint Programming International Conference*, volume 5202 de *LNCS*, pages 327–341. Springer-Verlag, 2008.

[20] Ferrante J. Cytron R., Rosen B. K., Wegman M. N. et Zadeck F. K. : Efficently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13, number 4:451–490, 1991.

[21] Rina Dechter : *Constraint Processing*. Morgan Kaufmann publisher, 2003.

[22] Thomson-CSF Detexis, LIFC, I3S, LSR et Axlog : INKA : Génération automatique déterministe de données de test selon des critères de couverture structurelle. `http://rntl04.irisa.fr/pres04/INKA.pdf`.

[23] Vijay D'Silva, Daniel Kroening et Georg Weissenbacher : A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[24] B. Dutertre et L. Mendonca de Moura : A fast linear-arithmetic solver for DPLL(T). *In CAV*, volume 4144 de *LNCS*, pages 81–94. Springer-Verlag, 2006.

[25] J.C. Filliâtre et Claude Marché : The why/krakatoa/caduceus platform for deductive program verification. *In CAV*, volume 4590 de *LNCS*, pages 173–177. Springer-Verlag, 2007.

[26] C. Flanagan : Automatic software model checking via constraint logic. *Science of Computer Programming*, 50 (1-3):253–270, 2004.

[27] Eclipse Foundation : Eclipse java development tools (JDT). `http://www.eclipse.org/jdt/`.

[28] A. Gotlieb : Génération structurelle de cas de test structurel avec la programmation par contraintes. Thèse de doctorat, Université de Nice-Sophia Antipolis.

[29] A. Gotlieb, B. Botella et M. Rueher : Automatic test data generation using constraint solving techniques. *In ISSTA'98*, volume 2, pages 53–62. ACM SIGSOFT, 1998.

[30] A. Gotlieb, B. Botella et M. Rueher : A CLP framework for computing structural test data. *In Computational Logic, CL'2000*, pages 399–413, 2000.

[31] F. Ivančić, Z. Yangb, M. K. Ganaia, A. Guptaa et P. Ashara : Efficient SAT-based bounded model checking for software verification, 2008.

[32] D. Jackson et M. Vaziri : Finding bugs with a constraint solver. *In ACM SIGSOFT Symposium on Software Testing and Analysis*, volume 25, pages 14–25, 2000.

[33] Gary T. Leavens et Yoonsik Cheon : JML home page. `http://www.cs.ucf.edu/~leavens/JML/`.

[34] LIFC, I3S, IMAG, THALES et Axlog : DANOCOPS : Détection automatique de non-conformités d'un programme vis à vis de sa spécification. `http://lifc.univ-fcomte.fr/danocops/`.

[35] A. Mackworth : Consistency in networks of relations. *Journal of Artificial Intelligence*, 8(1):99–118, 1977.

[36] A. Mackworth : Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[37] U. Montanari : Networks of constraints : Fundamental properties and applications to image processing. *Information science*, 7:95–132, 1974.

[38] Corina S. Pasareanu et Willem Visser : Verification of java programs using symbolic execution and invariant generation. *In SPIN*, volume 2989 de *LNCS*, pages 164–181. Springer-Verlag, 2004.

[39] Jean-Charles Régin : A filtering algorithm for constraints of difference in CSPs. *In AAAI*, pages 362–367, 1994, Seatle, Washington.

[40] Jean-Charles Régin : Modélisation et contraintes globales en programmation par contraintes. Habilitation à diriger des recherches, Université de Nice-Sophia Antipolis, Novembre 2004.

[41] N.T. Sy et Y. Deville : Automatic test data generation for programs with integer and float variables. *In 16th IEEE ASE*, 2001.

[42] Pascal Van Hentenryck : *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[43] Pascal Van Hentenryck et Laurent Michel : *Constraint-Based Local Search.* The MIT Press, 2005.

[44] Pugh W. : The omega test : A fast and practical integer programming algorithm for dependence analysis. *Communication fo the ACM*, 31(8), 1992.

Chapitre IV

Vérification des programmes basée sur la sémantique

Ce chapitre présente les travaux commencés pendant mon séjour à Cambridge dans l'équipe du professeur Mike Gordon. L'objectif est de fonder la vérification des programmes par exécution symbolique sur une sémantique opérationnelle du langage définie formellement dans l'assistant de preuves HOL4.

Je décris brièvement les principes d'HOL[1] et la philosophie spécifique aux assistants de preuves. Je détaille ensuite une première approche que nous avons implémentée dans HOL4 et qui reprend mes travaux sur la vérification des programmes par exécution symbolique (voir section III.4 page 112), mais qui base l'exécution symbolique sur une sémantique opérationnelle du langage définie formellement dans HOL4. Ce premier travail m'a permis de me familiariser avec HOL4 et a permis une première collaboration avec Mike Gordon sur les preuves par bounded model checking, alors que le plus souvent, avec un assistant de preuves, l'usage est de réaliser des preuves *complètes* en générant une *plus faible pré-condition*. Ce travail est intégré à la distribution actuelle de HOL4 (HOL4 est téléchargeable sur `http://hol.sourceforge.net/;nostravauxsetrouventdanslerpertoireexamples/opsemTools`) et a donné lieu à un rapport de recherche [4]. J'ai choisi de le présenter bien qu'il ne soit pas encore publié, car il soulève de nombreuses questions ouvertes et perspectives dont je discute dans le chapitre VI section VI.2. Enfin, je présente une réflexion de synthèse qui formalise les preuves "en avant" (principe utilisé dans le bounded model checking) sous la forme d'un ensemble de règles pour générer une *plus forte post-condition* et les compare aux preuves complètes qui produisent une *plus faible pré-condition*. Ce travail a donné lieu à une publication comme co-auteur pour célébrer le $75^{ième}$ anniversaire de Tony Hoare [7].

IV.1 L'assistant de preuves interactif HOL4

IV.1.1 Historique

Cette sous-section donne un bref aperçu de l'historique d'HOL. Mon objectif est d'en situer la philosophie de base et les majeures évolutions qui me paraissent significatives en tant que non spécialiste mais utilisatrice pour la vérification de programmes. L'article de Mike Gordon en l'honneur de Robin Milner [6] fournit un historique détaillé.

LCF HOL est un descendant du système LCF (Logic for Computable Functions), un programme de "vérification de preuves" créé à l'Université de Stanford par Robin Milner en 1972. La logique de LCF manipule des termes du lambda-calcul typé et des formules du calcul des prédicats. Ce système est décrit de la façon suivante par Robin Milner lui-même :

> "The proof-checking program is designed to allow the user interactively to generate formal proofs about computable functions and functionals over a variety of domains, including those of interest to the computer

[1] Dans tout ce chapître, j'emploierai le nom *HOL4* pour l'implémentation actuelle, et j'emploierai le nom plus générique *HOL* pour parler de la famille des assistants de preuves en logique d'ordre supérieur.

> scientist - for example, integers, lists and computer programs and their semantics. The user's task is alleviated by two features : a subgoaling facility and a powerful simplification mechanism".

Une preuve avec LCF consiste donc à travailler sur un *but* à prouver en le prouvant directement par des règles de simplification ou en appliquant des règles d'inférence pour générer des sous-buts.

Edimburg-LCF LCF présentait deux problèmes majeurs : la taille des preuves était limitée par la capacité mémoire, et le nombre de commandes pour manipuler les preuves était limité. LCF a donc donné lieu à une nouvelle version "Edimburg-LCF" qui présente deux améliorations majeures. D'une part, seuls les résultats des preuves (i.e. les *théorèmes*) sont sauvegardés. L'idée maîtresse est qu'il est inutile de garder le déroulement des preuves. Au contraire, les théorèmes sont typés par un *type abstrait de données* dont les valeurs prédéfinies sont les axiomes et les opérations sont les règles d'inférence. Ainsi, un nouveau théorème est correct par construction car il ne peut être obtenu qu'à partir d'une dérivation de règles d'inférence à partir d'axiomes. D'autre part, Edimburg-LCF est basé sur le méta-langage ML pour définir les règles d'inférence et les axiomes, ce qui rend le système extensible.

Cambridge-LCF Au début des années 1980, Larry Paulson et Gérard Huet ont apporté des améliorations, en particulier sur ML (ces travaux ont donné lieu au développement de Caml par l'équipe de G. Huet), sur la gestion mémoire, et la façon d'implémenter des outils de preuve. Ces travaux ont apporté une amélioration conséquente de la rapidité d'exécution de LCF.

HOL À la même période, Mike Gordon travaillait sur la vérification de matériel et a appliqué les idées de composition des agents CCS "Calculus of Communicating Systems" de Robin Milner à l'assemblage de composants matériels. Pour cela, il a défini la notation LSM (Logic of Sequential Machines), et l'a implémentée dans "Cambridge LCF-LSM". Comme la modélisation de matériel nécessite d'écrire des relations entre entrées/sorties du composant, les connexions internes étant cachées, et nécessite d'autre part de décrire des signaux qui sont des fonctions sur le temps, il s'est avéré indispensable d'avoir une logique d'ordre supérieur avec quantificateurs. HOL (pour Higher Order Logic) est donc une implémentation dans LCF d'une logique d'ordre supérieur. Les différentes évolutions de HOL sont présentées en détail dans [6].

Applications d'HOL N'étant pas une spécialiste de HOL, je ne donne pas ici un état de l'art exhaustif mais me contente de donner un aperçu des travaux que j'ai étudiés lors de mon séjour à Cambridge.

Le démonstrateur HOL et ses dérivés ont été appliqués à de nombreux domaines allant de la vérification réalisée complètement "à la main" de théorèmes mathématiques (e.g. la preuve de la consistance de l'axiome du choix que Lawrence C. Paulson a présentée lors de mon séjour à Cambridge [11]), à des preuves plus

automatisées comme la vérification de l'implémentation d'un Lisp sur différentes architectures [9].

HOL a été utilisé à son origine pour la vérification de matériel, avec les travaux précurseurs de Mike Gordon et Warren Hunt présentés section II.1.3 page 28. Ces travaux ont été poursuivis notamment pour formaliser en HOL le jeu d'instructions du processeur ARM6 ainsi que sa micro-architecture et vérifier la correction entre ces deux niveaux [5]. Les travaux plus récents de l'équipe de Mike Gordon portent sur la compilation de langages assembleur (voir section II.3 page 37).

Enfin, les travaux actuels concernent aussi des aspects plus théoriques, pour enrichir le cœur même d'HOL. Il s'agit en particulier de la formalisation en HOL de la "separation logic", une logique qui permet de gérer séparément deux états. Cela est utilisé en particulier pour modéliser la programmation objet en tenant compte du contexte d'appel des méthodes [13, 10]. D'autre part, de nombreux efforts portent sur l'intégration d'outils externes comme un solveur SAT [1, 14] ou un solveur SMT (travail en cours de Tjark Weber). Le principe est d'appeler le solveur externe pour son efficacité et d'utiliser des informations de la preuve effectuée pour reconstruire une preuve en HOL. Par exemple, pour un solveur SAT, si la formule est satisfiable, alors le solveur fournit un exemple qui sert à générer un théorème HOL grâce à la tactique "EXISTS_TAC" (voir exemple IV.2) ; si la formule n'est pas satisfiable on peut utiliser la dérivation des résolutions qui ont permis de générer la clause vide [14].

IV.1.2 Caractéristiques principales de HOL4

Je présente ici de façon succincte les idées de base pour le développement d'une preuve avec HOL4. Ceci est illustré avec l'exemple fourni dans le tutorial de la distribution HOL qui définit et montre des propriétés de la division euclidienne.

HOL encourage les définitions plutôt que les postulats d'axiomes comme dans LCF. En effet, la logique d'ordre supérieur rend possible la spécification de nombreux objets mathématiques uniquement à partir de définitions. Une session avec HOL4 se présente donc de la façon suivante. Tout d'abord il faut définir (ou utiliser) une *théorie* qui contient un ensemble de types, de signatures, d'axiomes et de théorèmes. Les *types* sont prédéfinis ou définis par l'utilisateur (de type Hol_type). Les *termes* sont des variables, des constantes, des applications de fonctions et des lambda-abstractions. Les théorèmes sont des termes particuliers de type thm, notés $hypothese \vdash conclusion$ (un axiome étant un théorème particulier où l'hypothèse est vide). L'exemple IV.1 présente une théorie définie pour prouver le théorème d'Euclide.

Exemple IV.1 (Théorie de la division euclidienne) La théorie de la division euclidienne utilise la théorie de l'arithmétique qui contient les types, définitions et propriétés élémentaires de l'arithmétique (e.g. le type num, la constante 0, les opérations élémentaires $+$, $-$, ... définies à partir de l'opérateur suc et 0, la propriété de commutativité du $+$, ...). Elle ajoute à cette théorie la définition de la divisibilité et de la primalité. À partir de ces définitions, et en utilisant différentes tactiques, des

syntaxe usuelle	syntaxe HOL
$\wedge$	/\
$\vee$	\/
$\neg$	~
$\forall$	!
$\exists$	?
$\Rightarrow$	==>
$\vdash$	|-
$\lambda x.$	\x.

FIG. IV.1 – Syntaxe des principaux opérateurs logiques en HOL

théorèmes sont prouvés et ajoutés à la théorie. Les lignes suivantes sont directement extraites du fichier d'exemple "euclid.sml" de la distribution de HOL ((*1*) est un commentaire). La syntaxe des principaux opérateurs HOL est donnée figure IV.1.

```
(*1*) open arithmeticTheory
(*2*) val divides_def =
   Define
      `divides a b = ?x. b = a * x`;
(*3*) val prime_def =
   Define
      `prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)`;
(*4*) val DIVIDES_0 = store_thm
   ("DIVIDES_0",
      ``!x. x divides 0``,
       METIS_TAC [divides_def,MULT_CLAUSES]);
```

La ligne (*1*) indique que la théorie d'Euclide utilise toutes les définitions et théorèmes de la théorie de l'arithmétique. La ligne (*2*) définit la division euclidienne sous le nom de "divides" avec la définition : a divise b si et seulement si il existe x tel que $b = a*x$. La ligne (*3*) définit la primalité d'un nombre p sous le nom de "prime" avec la définition que p est premier si et seulement si p est différent de 1 et pour tout x, si x divise p alors $x = 1$ ou $x = p$. La ligne (*4*) demande d'ajouter à la théorie le théorème de nom "DIVIDES_0" qui indique que pour tout x, x divise 0. Plus précisément, ce théorème est d'abord prouvé en appliquant la tactique "METIS" (qui combine plusieurs tactiques élémentaires et permet de prouver efficacement des propriétés de l'arithmétique), et en utilisant de façon spécifique la définition de "divides" et le théorème "MULT_CLAUSES" de la théorie de l'arithmétique :

```
|- !m n.  (0 * m = 0) /\ (m * 0 = 0) /\ (1 * m = m) /\ (m * 1 = m)
          /\ (SUC m * n = m * n + n) /\ (m * SUC n = m + m * n)
```

HOL est un *assistant de preuves interactif* c'est-à-dire qu'il aide l'utilisateur à effectuer la preuve en offrant des tactiques prédéfinies, en vérifiant que les théorèmes ou tactiques proposés s'appliquent effectivement, et en calculant le

Nom	Syntaxe	Théorème renvoyé
Assumption	`ASSUME t`	`t|- t`
Reflexivity	`REFL t`	`|- t=t`
Beta-conversion	`BETA_CONV (\x.t1)t2`	`|-(\x.t1)t2 = t1[t2/x]`
Substitution	`SUBST T1|-t1=t1',...,Tn|-tn=tn', T|-t(t1,...,tn)`	`T1...TN T|-t(t1',...,tn')`
Abstraction	`ABS x T|-t1=t2`	`T|-(\x.t1) = (\x.t2)`
Discharge	`DISCH t1 T|-t2`	`T-{t1}|-t1 ==> t2`
Modus Ponens	`MP T1|-t1 ==> t2, T2|-t1`	`T1 T2|-t2`

FIG. IV.2 – Règles d'inférence primitives

résultat. Il offre sept règles d'inférence primitives qui sont présentées dans la figure IV.2. Par exemple, la règle d'inférence "ASSUME" introduit le théorème t comme une hypothèse qui valide la conclusion t. L'utilisateur peut définir lui-même sa propre tactique comme un assemblage de tactiques primitives ou de tactiques pré-définies et en indiquant quels théorèmes utiliser.

Le déroulement d'une preuve en HOL est de deux types : les preuves en avant qui utilisent principalement les règles d'inférence "ASSUME", "MP" et "DISCH" (de telles preuves sont en général assez délicates à mener car peu intuitives), et les preuves dirigées par des *buts*. Dans ce dernier cas, les preuves sont des successions d'utilisation de règles d'inférence primitives ou pré-définies. À chaque instant, l'utilisateur indique quelle tactique il veut utiliser. HOL4 vérifie que la tactique s'applique et calcule les sous-buts à prouver pour finir la preuve. Le processus est rendu plus efficace grâce à un ensemble de procédures de décision associées à certaines théories (e.g. vérification de tautologies basée sur la représentation des BDDs). De plus, la recherche de la preuve peut être dissociée de la preuve elle-même. Des programmes externes (comme un solveur SAT par exemple) peuvent être utilisés comme des oracles pour fournir le résultat d'une preuve, sans que cette preuve ait besoin d'être construite en HOL (dans ce cas, le théorème résultat est "étiqueté" comme oracle pour que l'on sache qu'il n'a pas été prouvé avec la logique d'HOL). L'exemple IV.2 montre la preuve d'un lemme pour théorème d'Euclide.

Exemple IV.2 (Preuve d'un lemme pour le théorème d'Euclide) La preuve du théorème d'Euclide est assez longue et complexe et peut être trouvée dans la distribution HOL et dans l'annexe VII.2.1 page 271. Le théorème d'Euclide qui dit que l'ensemble des nombres premiers est infini (i.e. `!n. ?p. n < p ∧ prime p`) est prouvé par induction sur n, il est donc nécessaire de prouver le lemme intermédiaire : `!x. divides x 0` pour le cas de base. Ce lemme peut être prouvé directement grâce à la tactique "METIS_TAC" comme dans l'exemple IV.1. Nous montrons ici ce qui se passe si l'on utilise uniquement des tactiques primitives. Pour être plus précise et montrer la pile des buts, je donne une copie textuelle de l'interaction avec HOL4. Tout d'abord, on donne le but à prouver avec la commande "g".

- g '!x. divides x 0';

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
  Initial goal:
  !x. divides x 0
  : proofs
```

Il y a une preuve en cours et un but à prouver. On demande maintenant à "expanser" (commande e) la tactique de réécriture "RW_TAC" en utilisant l'ensemble de règles de simplification de l'arithmétique "arith_ss" et en utilisant la définition de la division "[divides_def]".

```
- e (RW_TAC arith_ss [divides_def]);
OK..
  1 subgoal:
  > val it =  ?x'. (x = 0) ∨ (x' = 0)
```

Lors de l'expansion de la tactique, la définition de "divides" a été appliquée avec $a = x$ et $b = 0$ pour obtenir `divides x 0 =?x'.0=x*x'` et le théorème `MULT_EQ_0 = |- !m n.(m*n=0) = (m=0) ∨ (n=0)` a été utilisé comme règle de réécriture. Il reste alors à prouver le sous-but `?x'. (x = 0) ∨ (x' = 0)`. Ce but est prouvé de façon triviale si l'on instancie x' avec 0 ; il suffit pour cela d'utiliser la tactique "EXISTS_TAC".

```
- e (EXISTS_TAC ``0``);
OK..
  1 subgoal:
  > val it =  (x = 0) ∨ (0 = 0)
- e (RW_TAC arith_ss []);
OK..
  Goal proved.
  |- (x = 0) ∨ (0 = 0)
  Goal proved.
  |- ?x'. (x = 0) ∨ (x' = 0)
> val it = Initial goal proved.
|- !x. divides x 0
```

Ceci termine la preuve du lemme. Évidemment, même si des tactiques comme la tactique "METIS_TAC" permettent de prouver un grand nombre de formules de l'arithmétique sans autre précision, mener une preuve avec HOL4 requiert une grande habitude et une bonne connaissance des tactiques et théorèmes existants.♯

Règle de simplification pour la vérification de programmes Je termine cette sous-section introductive à HOL4 par deux exemples de règles que nous avons définies dans le cadre de la vérification des programmes Java par rapport à une spécification JML. L'exemple IV.3 illustre une tactique définie pour calculer la négation de la spécification JML et l'exemple IV.4 illustre comment transformer une quantification universelle sur un domaine fini en une conjonction. Ce dernier exemple illustre aussi le fait qu'HOL4 a une logique d'ordre supérieur : la tactique s'applique à un prédicat.

Exemple IV.3 (Négation d'une spécification JML) Bon nombre des spécifications JML des programmes que nous avons vérifiés sont en fait une conjonction des cas d'utilisation de la méthode. Par exemple, pour le programme *Tritype* de classification d'un triangle en fonction de trois entrées i, j et k (voir figure III.14 page 123), la spécification JML est :

```
/*@ requires (i>=0)&&(j>=0)&&(k>=0);
 @ ensures
 @   ((i+j<=k)||(j+k<=i)||(i+k<=j)) ==> \result == 4 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&((i==j)&&(j==k)) ==> \result == 3 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&((i==j)||(j==k)||(i==k)) ==> \result == 2 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&!((i==j)||(j==k)||(i==k)) ==> \result == 1;
@*/
```

Pour utiliser la programmation par contraintes comme une procédure de décision, nous prenons la négation de la spécification. Pour calculer efficacement cette négation, tout en gardant une forme qui reste proche de la spécification initiale (et on peut l'espérer des expressions du programme), nous avons défini une règle de conversion[2] qui utilise la loi de De Morgan à un seul niveau. La règle de conversion est la suivante :

```
NOT_CONJ_IMP_CONV ¬((A1 ==> B1) ∧ ... ∧ (An ==> Bn) ∧ TM) =
  |- (A1 ∧ ¬B1) ∨ ... ∨ (An ∧ ¬Bn) ∨ ¬TM
```

Elle a été écrite en ML, et utilise des règles de conversion et des théorèmes basiques comme par exemple le théorème de la théorie des booléens qui énonce la loi de De Morgan :

```
DE_MORGAN_AND_THM = !A B. ¬(A ∧ B) = ¬A ∨ ¬B
```

Dans cette règle de conversion les Ai et Bi sont des termes quelconques.♯

Exemple IV.4 (Quantificateur universel borné) La primitive \forall de JML permet d'exprimer une propriété pour un ensemble de valeurs comprises entre deux bornes. Par exemple, la pré-condition de la recherche binaire d'une valeur dans un tableau en JML est la suivante :

```
\forall int i ;0<=i && i<a.length-1 ; a[i]<=a[i+1]
```

La règle de conversion `boundedForAll_CONV`, écrite en ML, utilise des règles de conversion basiques et le théorème d'ordre supérieur suivant :

```
BOUNDED_FORALL_THM = |- !c. 0<c ==> (( !n. n<c ==> P n) = P (c-1) ∧
  !n. n<c-1 ==> P n) : thm
```

Ce théorème expanse d'un cran la propriété $P(n)$ vraie pour tout n inférieur à c. P est ici un prédicat quelconque qui s'applique à la variable n ; c'est en ce sens que la règle de conversion est d'ordre supérieur.

Cette règle de conversion simplifie au passage les sous-termes obtenus. Appliquée à une fonction f quelconque, elle calcule simplement la conjonction de tous les $f(i)$ pour $i < c$.

[2]Une règle de conversion est un théorème qui est utilisé par une tactique de réécriture de gauche à droite.

```
-val tm = ‘‘ !n. n<8 ==> f(n)‘‘ ;
-val btm = boundedForAll_CONV tm ;
> val btm = |- ( !n. n<8 ==> f n) =
   f 7 ∧ f 6 ∧ f 5 ∧ f 4 ∧ f 3 ∧ f 2 ∧ f 1 ∧ f 0 : thm
```

Si maintenant f est le prédicat $n < n + 1$ qui est toujours vrai, la règle de conversion renvoie T c'est-à-dire la valeur "vraie".

```
-val tm2 = ‘‘ !n. n < 8 ==> n<n+1‘‘ ;
-val btm2 = boundedForAll_CONV tm2
> val btm2 = |- ( !n. n<8 ==> n<n+1) = T : thm  ♯
```

IV.2 Vérification des programmes par exécution symbolique basée sur la sémantique

Je présente ici mes travaux sur le bounded model checking (BMC) de programmes dont la sémantique est définie en logique d'ordre supérieur. L'idée initiale est de trouver un compromis entre l'efficacité fournie par des solveurs externes et une plus grande correction en basant les étapes de l'exécution symbolique sur la sémantique définie dans HOL4. Notre objectif à long terme est d'offrir aussi bien une preuve incomplète par BMC qu'une preuve complète avec HOL4 dans un cadre sémantique cohérent.

IV.2.1 Principes

Le principe général est celui présenté dans l'algorithme section III.4 figure III.13 : le programme est exécuté symboliquement en coupant à la volée les chemins infaisables. Une première différence est la représentation de l'état. L'état symbolique du programme est ici une liste associative qui lie le nom des variables à leur valeur courante[3]. Ce n'est donc pas le système de contraintes qui détermine la valeur de l'état. D'autre part, le calcul de l'état après exécution d'une instruction ainsi que la preuve en fin de chemin font appel à des solveurs externes et à HOL4. Plus précisément :

(i) Pour chaque étape d'exécution symbolique, le nouvel état est calculé par une fonction de transition d'état dérivée automatiquement par le démonstrateur de théorèmes à partir de la sémantique formelle appliquée au programme.

(ii) Quand une instruction conditionnelle est atteinte, les solveurs externes (un solveur SMT pour les expressions linéaires et un solveur de contraintes pour les expressions non linéaires) sont appelés pour savoir si l'état détermine la condition. Si c'est le cas, un seul chemin est exploré, sinon les deux branches le sont.

(iii) À la fin de chaque chemin, une preuve déductive avec HOL4 puis avec les solveurs externes est essayée en séquence. Si HOL4 échoue à vérifier la formule, alors les solveurs fournissent une procédure de décision pour un sous-ensemble des entiers.

[3] Voir section IV.5.1 pour la formalisation de l'état par une telle association

(iv) Les théorèmes prouvés avec les solveurs externes sont étiquetés pour indiquer comment le théorème a été prouvé et l'étiquette est propagée chaque fois que le théorème est utilisé.

Par rapport à notre approche précédente, le point (i) calcule l'état par réduction automatique de théorèmes prouvés à partir de la sémantique formelle du langage d'entrée. Ainsi les formules obtenues à la fin des chemins portent sur des états qui sont sémantiquement corrects. Le point (ii) est similaire mais utilise conjointement un solveur SMT et un solveur de contraintes. Ceci est motivé par des raisons pratiques. Les solveurs SMT sont très efficaces pour résoudre des formules linéaires et il existe une distribution libre et donc intégrable à la distribution HOL4 de tels solveurs [4]. Par contre, le solveur de contraintes d'Ilog qui m'est familier n'est pas libre ; il est donc utilisé en dernier recours dans le cas non linéaire que le solveur SMT ne peut pas traiter. Ainsi, seuls les exemples non linéaires nécessitent l'accès à une licence fournie par Ilog[5]. Le point (iii) combine HOL4 et les solveurs externes. En particulier, HOL4 fournit des outils de simplification de haut niveau (comme par exemple les règles de conversion des exemples IV.3 et IV.4), qui assurent la correction des transformations qui étaient faites directement en Java dans la version précédente. Enfin le point (iv) permet de savoir à chaque instant si la vérification effectuée jusqu'alors est due à HOL4 ou aux solveurs externes. Ceci respecte la philosophie d'HOL où tout théorème doit être une dérivation à partir de définitions et de théorèmes initiaux par des règles d'inférence. Si un solveur externe a été utilisé comme oracle, alors cette chaîne n'est plus garantie et il faut le signifier.

Dans la suite de cette section, je présente les points marquants de cette nouvelle approche : la définition de la sémantique formelle et l'appel des solveurs depuis HOL4. L'algorithme d'exécution symbolique est délibérément laissé de côté car très similaire à celui de l'approche précédente. Je discute brièvement de résultats expérimentaux et j'énumère enfin les réalisations logicielles qui ont été nécessaires ; cela m'a pris beaucoup de temps.

IV.2.2 Sémantique opérationnelle

Les programmes et leurs spécifications sont représentés par des termes en logique d'ordre supérieur. Les figures IV.3 et IV.4 présentent respectivement un programme simple qui calcule la valeur absolue de ses entrées[6] et sa représentation sous forme de termes HOL4[7]. Il s'agit d'une spécification relationnelle entre la lambda expression de la pré-condition (qui renvoie T ici), le programme dans la syntaxe du langage d'entrée (`Seq` est une séquence de deux instructions) et la lambda expression de la post-condition qui donne l'état final en fonction de l'état initial ($s\hat{}v$ est la valeur de la variable v dans l'état s).

[4]Plus précisément j'ai utilisé le solveur SMT Yices `http://yices.csl.sri.com/`.

[5]Il existe des solveurs de contraintes libres, les solveurs Gecode `http://www.gecode.org/index.html` et COMET `http://www.comet-online.org/Welcome.html` étant les plus complets et performants. Je n'ai pas eu le temps d'écrire une version pour ces solveurs.

[6]Cet exemple a déjà été présenté section III.4 page 112

[7]Dans la suite de la section, contrairement aux exemples présentés précédemment, nous n'utiliserons pas la syntaxe concrète d'HOL4 mais une syntaxe logique usuelle.

```
1  class AbsMinus {
2  /*@ ensures   ((i < j) ==> (\result == j-i))
3             && ((i >= j) ==> (\result == i-j));  @*/
4  int absMinus (int i, int j) {
5    int result;
6    int k = 0;
7    if (i <= j) k = k+1;
8    if (k == 1 && i != j) result = j-i;
9       else result = i-j;  // ERROR: result = j-i;
10   return result;}}
```

FIG. IV.3 – Programme AbsMinus en Java

```
1  RSPEC
2   (λs. T)
3   (Seq (Assign "result" (Const 0))
4        (Seq (Assign "k" (Const 0))
5             (Seq (Cond
6                     (LessEq  (Var "i") (Var "j"))
7                     (Assign "k" (Plus (Var "k") (Const 1)))
8                     Skip)
9                  (Seq (Cond
10                          (And (Equal (Var "k") (Const 1))
11                               (Not (Equal (Var "i") (Var "j"))))
12                          (Assign "result" (Sub (Var "j") (Var "i")))
13                          (Assign "result" (Sub (Var "i") (Var "j"))))
14                       (Assign "Result" (Var "result"))))))
15  (λs1 s2. (s1^"i" < s1^"j"  ==> s2^"Result" = s1^"j" - s1^"i") ∧
16           (s1^"i" >= s1^"j" ==> s2^"Result" = s1^"i" - s1^"j"))
```

FIG. IV.4 – Spécification relationnelle en HOL du programme AbsMinus

La sémantique opérationnelle d'un petit langage de programmation impératif a été définie ; elle relie les instructions du langage aux opérateurs de base d'HOL4. La fonction `EVAL i s1 s2` détermine l'état `s2` obtenu après exécution de l'instruction `i` sur l'état courant `s1` (voir figure IV.6). Par exemple, la sémantique d'une instruction conditionnelle `(Cond b i1 i2)` est la suivante :

```
(∀i1 i2 s1 s2 b. (EVAL i1 s1 s2 ∧ beval b s1)⇒EVAL (Cond b i1 i2) s1 s2)∧
(∀i1 i2 s1 s2 b. (EVAL i2 s1 s2 ∧ ¬(beval b s1))⇒EVAL (Cond b i1 i2) s1 s2)
```

La première ligne indique que si l'exécution de l'instruction `i1` sur l'état `s1` donne l'état `s2`, et que l'évaluation de la condition `b` sur `s1` est vraie, alors l'état résultant de l'exécution de la conditionnelle sur l'état `s1` est `s2`. La deuxième ligne indique que si l'évaluation de la condition `b` donne la valeur faux alors le résultat est l'état après exécution de `i2`. Dans cette définition, *beval* est la sémantique des opérations booléennes (voir figure IV.5). Par exemple, la sémantique de l'opérateur *Or* est la suivante :

```
∀ b1 b2 s. beval (Or b1 b2) s = (beval b1 s)  ∨ (beval b2 s)
```

Afin d'effectuer l'exécution symbolique, il a été nécessaire de définir une sémantique "small-step" pour pouvoir exécuter pas à pas chaque instruction d'un programme. C'est le rôle de la fonction `STEP1` : si `STEP1(l1,s1) = (l2,r)` alors exécuter une étape de l'instruction en tête de la liste `l1` dans l'état `s1` renvoie `(l2,r)`, où `l2`

```
(∀ e1 e2 s. beval (Equal e1 e2) s = (neval e1 s = neval e2 s)) ∧
(∀ e1 e2 s. beval (Less e1 e2) s = (neval e1 s) < (neval e2 s)) ∧
(∀ e1 e2 s. beval (LessEq e1 e2) s = (neval e1 s)≤ (neval e2 s)) ∧
(∀ b1 b2 s. beval (And b1 b2) s = (beval b1 s ∧ beval b2 s)) ∧
(∀ b1 b2 s. beval (Or b1 b2) s = (beval b1 s ∨ beval b2 s)) ∧
(∀ b s. beval (Not b) s = ¬(beval b s))
```

FIG. IV.5 – Sémantique des expressions booléennes *(`beval` définit la sémantique des expressions booléennes, `neval` définit la sémantique des expressions numériques)*

est le reste des instructions à exécuter et `r` est le résultat. `r` peut être soit `RESULT(s2)` si l'étape s'est bien passée ou `ERROR(s2)` si une assertion a échoué. La figure IV.7 présente cette sémantique, qui a été prouvée équivalente à la sémantique "big step" de la figure IV.6 dans HOL4.

Lors de l'exécution symbolique, les fonctions `STEP1` et `beval` sont calculées efficacement dans HOL4 grâce à un mécanisme de réduction dû à Barras [2]. Plus précisément, nous avons implémenté trois fonctions : "`nextState i s`" renvoie l'état après exécution de l'instruction i sur l'état s si le résultat est de type `RESULT` et lève une exception sinon, "`nextInst c s`" renvoie la liste des instructions à exécuter et "`evalCond b s`" renvoie la valeur du terme b sur l'état s. Les fonctions `nextState` et `nextInst` exécutent `STEP1` par le mécanisme de réduction tandis que `evalCond` exécute `beval` par ce même mécanisme. Par exemple, pour le programme `AbsMinus` de la figure IV.4, quand `b` est le terme

```
(And (Equal (Var "k") (Const 1)) (Not (Equal (Var "i") (Var "j"))))
```

et que la valeur de l'état courant s associe la constante 1 à la variable k, `evalCond b s` exécute `beval b s` avec le mécanisme de réduction pour obtenir le théorème :

```
⊢ beval b s = (Not (Equal (Var "i") (Var "j")))
```

et renvoyer la partie droite de la conclusion de ce théorème.

Ces trois fonctions sont utilisées dans l'algorithme d'exécution symbolique : `nextState` et `nextInst` pour calculer l'état suivant et le reste des instructions à exécuter et `evalCond` pour simplifier la condition d'une instruction conditionnelle et savoir si elle se réduit trivialement à T ou F.

IV.2.3 Appels de solveurs externes depuis HOL4

Le système HOL4 a de nombreuses fonctions pré-définies pour prouver des théorèmes (e.g. différentes procédures de décision et solveurs du premier ordre). Nous avons ajouté deux fonctions ML : `extSMTSolv tm to` pour appeler un solveur SMT et `extCPSolv tm to f` pour appeler un solveur de contraintes. Le premier argument `tm` est un terme existentiellement quantifié dont on veut tester la satisfiabilité, et le second argument, `to` est un "timeout". Le troisième argument `f` est nécessaire pour fixer le domaine des variables dans le système de contraintes. La fonction `extSMTSolv` est appelée quand `tm` est linéaire et la fonction `extCPSolv` quand il n'est pas linéaire.

Quand le solveur externe trouve une solution (i.e. `tm` est satisfiable) alors ces fonctions renvoient un théorème qui est automatiquement généré par la tactique "EXISTS_TAC" avec la valeur fournie par le solveur. S'il n'y a pas de solution,

```
   (∀s. EVAL Skip s s)
 ∧ (∀s v e. EVAL (Assign v e) s (s+(v,(neval e s))))
 ∧ (∀s v. EVAL (Dispose v) s (s-v))
 ∧ (∀i1 i2 s1 s2 s3. EVAL i1 s1 s2 ∧ EVAL i2 s2 s3 ⇒ EVAL (Seq i1 i2) s1 s3)
 ∧ (∀i1 i2 s1 s2 b.  EVAL i1 s1 s2 ∧ beval b s1 ⇒ EVAL (Cond b i1 i2) s1 s2)
 ∧ (∀i1 i2 s1 s2 b. EVAL i2 s1 s2 ∧ ¬(beval b s1) ⇒ EVAL (Cond b i1 i2) s1 s2)
 ∧ (∀i s b. ¬beval b s ⇒ EVAL (While b i) s s)
 ∧ (∀i s1 s2 s3 b. EVAL i s1 s2 ∧ EVAL (While b i) s2 s3 ∧ beval b s1 ⇒
       EVAL (While b i) s1 s3)
 ∧ (∀i s1 s2 v. EVAL i s1 s2 ⇒ EVAL (Local v i) s1
       (if v ∈ s1 then s2+(v,(s1^v)) else s2-v))
```

FIG. IV.6 – Sémantique "big-step" *(s+(v,n) est l'état obtenu à partir de s en donnant la valeur n à la variable v ; s^v est la valeur de v dans s, v ∈ s signifie que v est défini dans s et s-v est le résultat de la suppression de v dans s)*

```
  (STEP1([], s) = ([], ERROR s))
∧ (STEP1(Skip::l, s) = (l, RESULT s))
∧ (STEP1(Assign v e::l, s) = (l, RESULT(s+(v,(neval e s)))))
∧ (STEP1(Dispose v::l, s) = (l, RESULT(s-v)))
∧ (STEP1(Seq i1 i2::l, s) = (i1::i2::l, RESULT(s)))
∧ (STEP1(Cond b i1 i2::l, s) = if beval b s then (i1::l, RESULT s)
       else (i2::l, RESULT s))
∧ (STEP1(While b i::l, s) = if beval b s then (i::While b i::l, RESULT s)
       else (l, RESULT s))
∧ (STEP1(Local v i::l, s) = ifv∈sthen(i::Assign v (Const(s^v))::l,RESULTs)
       else(i::Dispose v::l, RESULT s))
```

FIG. IV.7 – Sémantique "small-step" *(: : est l'opérateur "cons" des listes)*

alors ces fonctions renvoient un *théorème étiqueté* où l'étiquette indique comment le théorème a été prouvé. Une chaîne de caractères identifie le solveur utilisé, et dans le cas du solveur de contraintes, cette chaîne contient aussi le format des entiers.

Les fonctions `extCPSolv` et `extSMTSolv` servent dans l'algorithme d'exécution symbolique pour tester la faisabilité d'un chemin et sa correction (voir fonctions $test_{cond}$ et $test_{chemin}$ de l'algorithme figure III.13 page 115). Le test des conditions des chemins est répété très souvent. L'accent doit donc être mis sur l'efficacité. Pour cela, la condition est d'abord évaluée sur l'état courant grâce à la fonction `evalCond` (ceci est rapide grâce au mécanisme de réduction). Si le terme simplifié obtenu b se réduit à `T` ou `F`, seule la branche correspondante est explorée. Sinon, les solveurs externes sont appelés avec un petit "timeout" pour tester la satisfiabilité de la conjonction de b, de la pré-condition, et des décisions qui ont été prises sur le chemin courant. Si le "timeout" est atteint dans les solveurs, alors les deux branches sont explorées[8].

Tandis que l'efficacité est le point clé pour tester les chemins, la robustesse est une

[8]Dans ce cas, on explore au pire des cas un des deux chemins pour rien. Notons que le timeout n'a pas été atteint dans les exemples que nous avons traités.

priorité pour vérifier leur correction. Par conséquent, le démonstrateur de théorèmes est appelé en premier. Comme chaque chemin est vérifié séparément, les formules à vérifier ne contiennent pas de conditionnelles et sont relativement petites ; il est ainsi possible d'appliquer une preuve déductive avec HOL4 (qui n'est pas prévu pour manipuler des énormes formules avec une structure booléenne complexe contrairement aux solveurs SAT ou SMT généralement employés dans les outils de BMC). Les solveurs externes ne sont appelés que dans le cas où le démonstrateur de théorèmes échoue. Un grand "timeout" est utilisé (et un grand format pour les entiers avec le solveur de contraintes). Si le solveur externe réussit à effectuer la preuve alors un théorème étiqueté est retourné. Par contre, si le délai est atteint, ou si une assurance plus élevée que celle fournie par le solveur est souhaitée, une preuve interactive peut être commencée dans HOL4.

IV.2.4 Résultats expérimentaux

Cette nouvelle approche a été appliquée à un sous-ensemble significatif des exemples traités section III.4.5 page 119, notamment le programme *Tritype*, la recherche binaire d'un élément dans un tableau, le tri à bulles avec condition initiale et la somme des carrés des entiers de 1 à n et de p à n (ce cas étant plus difficile car la valeur de p est quelconque et donc la somme ne se réduit pas à une constante).

Les résultats expérimentaux ont prouvé que cette approche est faisable, même si elle est, comme attendue, plus lente que la précédente : de l'ordre de 200 fois plus lente en moyenne que l'approche par exécution symbolique avec les contraintes. Tous les exemples, excepté les non linéaires (i.e. somme des carrés), ont été vérifiés avec HOL4, et environ la moitié des conditions de chemin ont été décidées avec HOL4 (i.e. réduites à T ou F). Le solveur SMT a permis de décider les autres conditions de façon très efficace (un appel au solveur SMT sur ce type de formules prend de l'ordre de $0,01s$). Le solveur de contraintes a servi aux cas non linéaires.

Cependant, si les réductions mécaniques effectuées par HOL4 fondent l'exécution symbolique sur la sémantique du langage d'entrée, elles ont malheureusement un fort impact sur les performances. L'exécution de la sémantique (i.e. les fonctions `nextState`, `nextInst` et `evalCond`) est rapide mais peut devenir lente si la taille des formules augmente (par exemple plus de variables d'état ou des termes plus grands). Le temps d'exécution pour `evalCond` varie de $0.054s$ pour *Tritype* à $3.576s$ pour *BubbleSort*, et le temps d'exécution pour `nextState` varie de $0.058s$ pour la somme à $5.336s$ pour *BubbleSort*. La fonction `nextState` est appelée très souvent et a donc un fort impact sur les temps d'exécution. En outre, les procédures de décision de HOL4 employées pour vérifier la correction des chemins sont appelées rarement mais sont très lentes, de $3.340s$ pour *Tritype* à $96.846s$ pour *BubbleSort*.

IV.2.5 Réalisations logicielles

Le travail présenté ci-dessus a été réalisé en étroite collaboration avec Mike Gordon qui m'a fourni des exemples en ML et HOL4 pour que je puisse démarrer plus facilement. Il a également écrit la sémantique du langage en HOL4 et effectué la preuve d'équivalence entre la version "small-step" et "big-step" de cette sémantique.

Enfin, il m'a conseillée en permanence en m'indiquant quelles tactiques ou procédures de décision utiliser, et a écrit certains des outils de simplification.

Pour ma part, j'ai effectué un grand nombre de réalisations logicielles qui sont livrées dans la distribution actuelle de HOL4 (répertoire examples/opsemTools). Tout d'abord, afin de pouvoir tester l'approche sur des exemples de taille significative, j'ai écrit (en Java avec JDT[9]) un traducteur qui prend un programme Java et une spécification JML et génère une spécification relationnelle dans la syntaxe HOL4. Il m'a fallu ensuite gérer l'appel aux solveurs externes depuis HOL4. Ces modules sont écrits en ML. Ils prennent en entrée un terme HOL4 et génère l'entrée vers le solveur SMT Yices ou vers le solveur de contraintes JSolver. Enfin, j'ai écrit le module d'exécution symbolique en ML. Il fait appel à la sémantique définie en HOL4, à différents outils de simplification spécifiques écrits en HOL4 et enfin aux solveurs externes. Il s'agit d'une chaîne complètement automatique : la fonction ML $verify$ prend en entrée le nom d'un programme à vérifier, génère la représentation interne, fait l'exécution symbolique du programme en fournissant une trace des chemins explorés et renvoie le terme $RESULT\ s$ si le programme est correct (s est l'état final), et renvoie $ERROR\ s$ s'il y a une erreur. Dans ce dernier cas, s est instancié avec une des erreurs (i.e. solution trouvée par le solveur externe).

J'effectue une critique de ce premier travail dans la section VI.2 et en présente les questions ouvertes et perspectives. La section suivante est une formalisation de l'exécution symbolique en terme de génération de plus forte post-condition.

IV.3 Vérification des programmes par génération d'une plus forte post-condition

Cette section expose quelques éléments de l'article que Mike Gordon a été invité à écrire pour célébrer le $75^{ième}$ anniversaire de Tony Hoare, et dont je suis co-auteur. Il est inclus comme article support section IV.5.1 page 186.

J'introduis tout d'abord, de façon succincte et sans aucune perspective historique, les preuves par génération d'une plus faible pré-condition, et détaille l'exécution symbolique vue comme une génération de plus forte post-condition. Ce dernier point fournit un cadre théorique aux travaux de la section précédente. De plus, il pose de nombreuses questions et perpectives qui seront discutées au chapitre VI.

IV.3.1 Vérification de triplets de Hoare

Un *triplet de Hoare* $\{P\}S\{Q\}$ (où P et Q sont des assertions et S est un ensembles d'instructions) signifie que Q (la "post-condition") est vraie dans chaque état atteint par l'exécution de S à partir d'un état initial dans lequel P (la "pré-condition") est vraie. La logique de Hoare [8] est un système déductif dont les axiomes et les règles d'inférence fournissent une méthode pour prouver de tels triplets.

La génération de *plus forte post-condition* et *plus faible pré-condition* permet également de prouver de tels triplets. Si $P \Rightarrow Q$ alors P est dit *plus fort* que

[9]Eclipse Java development tools (JDT) `http://www.eclipse.org/jdt/`.

Q. Un générateur[10] de *plus forte post-condition* pour un programme S transforme le prédicat de pré-condition P en un prédicat de post-condition $\mathtt{sp}\,S\,P$, qui est le prédicat 'le plus fort' qui est vrai après avoir exécuté S dans un état qui satisfait la pré-condition P. Il est plus fort dans le sens que si Q est vrai pour chaque état résultant de l'exécution de S quand P est vrai, alors $\mathtt{sp}\,S\,P$ est plus fort que Q – i.e. $\mathtt{sp}\,S\,P \Rightarrow Q$.

Si $P \Rightarrow Q$ alors Q est *plus faible* que P. Un générateur de *plus faible pré-condition* pour un programme S transforme le prédicat de post-condition Q en un prédicat de pré-condition $\mathtt{wp}\,S\,Q$, qui est le prédicat le 'plus faible' qui assure que si un état le satisfait alors après exécution de S le prédicat Q est vrai. Il est plus faible dans le sens que si P a la propriété qu'exécuter S quand P est vrai assure que Q est vrai après exécution, alors $\mathtt{wp}\,S\,Q$ est plus faible que P – i.e. $P \Rightarrow \mathtt{wp}\,S\,Q$.

Le lien entre ces deux générateurs et la vérification d'un triplet de Hoare est que le triplet $\{P\}\,S\,\{Q\}$ est vérifié si et seulement si $(\mathtt{sp}\,S\,P) \Rightarrow Q$ et aussi si et seulement si $P \Rightarrow \mathtt{wp}\,S\,Q$. Les équations satisfaites par $\mathtt{sp}\,S\,P$ et $\mathtt{wp}\,S\,Q$ sont données figure IV.8. Calculer $\mathtt{sp}\,(S_1\,\mathtt{;}\,S_2)\,P$ avec ces équations revient à partir d'une pré-condition P, à calculer d'abord $\mathtt{sp}\,S_1\,P$ et ensuite appliquer $\mathtt{sp}\,S_2$ au prédicat résultant. Cela correspond à une exécution symbolique en avant. Au contraire, calculer $\mathtt{wp}\,(S_1\,\mathtt{;}\,S_2)\,Q$ procède en arrière à partir de la post-condition Q en calculant d'abord $\mathtt{wp}\,S_2\,Q$ en appliquant ensuite $\mathtt{wp}\ S_1$ au prédicat résultant.

Une différence majeure entre les deux approches réside dans la règle de transformation de l'affectation. Le fait que cette règle soit plus complexe dans le cas en avant (à cause du quantificateur existentiel), semble expliquer pourquoi la plupart des prouveurs utilisent la génération en arrière. En effet, la règle de génération d'une plus faible pré-condition pour la sémantique de l'affectation $V\,\mathtt{:=}\,E$ est :

$\mathtt{wp}\,(V\,\mathtt{:=}\,E)\,Q \;=\; Q[E/V]$

tandis que la règle de génération d'une plus forte post-condition est :

$\mathtt{sp}\,(V\,\mathtt{:=}\,E)\,P \;=\; \exists v.\ (V = E[v/V]) \wedge P[v/V]$

où la notation $M[E/V]$, avec M formule ou expression, représente le résultat de la substitution de V par E dans M. Pour la plus faible pré-condition, il suffit de remplacer V par E dans Q. Pour la plus forte post-condition, des quantificateurs existentiels sont introduits afin de représenter les modifications successives de la variable. En effet, $\exists v.\ (V = E[v/V]) \wedge P[v/V]$ signifie qu'il existe une variable v qui était la valeur de V avant exécution de l'affectation et qui vérifie la pré-condition. De plus, la variable V après affectation (i.e. partie gauche de $V = E[v/V]$) est calculée en fonction de cette valeur précédente. Notons que les renommages SSA utilisés dans notre précédente approche peuvent être vus comme une skolémisation de ces quantificateurs existentiels[11]. L'exemple IV.5 compare la génération d'une plus faible pré-condition, d'une plus forte post-condition et de la forme SSA pour vérifier un triplet de Hoare très simple.

Exemple IV.5 (Régle d'affectation) Soit le triplet de Hoare suivant :
{x=0}x :=x+1 ;x :=x+2 ;{x=3}, spécifiant le programme nommé "TresSimple".

[10] J'ai choisi d'appeler "générateur de plus forte post-condition" la notion de "strongest postcondition predicate transformer".

[11] Cette équivalence n'a pas encore pu être prouvée formellement avec HOL4.

Plus faible pré-condition
La plus faible pré-condition est générée en partant de la post-condition, en procédant vers l'arrière depuis la dernière instruction du programme et en remplaçant dans la post-condition courante les variables par les valeurs qui leur sont affectées.

$\texttt{wpTresSimple}\ (\texttt{x} = 3) =$
$\quad \texttt{wp}(\texttt{x} := \texttt{x} + 1; \texttt{x} := \texttt{x} + 2;)\quad (\texttt{x} = 3) =$
$\quad \texttt{wp}(\texttt{x} := \texttt{x} + 1;)\quad (\texttt{x+2} = 3) =$
$\quad (\texttt{(x+1)+2} = 3)$

Pour montrer le triplet de Hoare {x=0}x :=x+1 ;x :=x+2 ;{x=3} il faut alors montrer que $\texttt{x=0} \Rightarrow \texttt{wpTresSimple(x=3)}$ c'est-à-dire que $\texttt{x=0} \Rightarrow \texttt{(x+1)+2=3}$ ce qui est trivialement vérifié.
Plus forte post-condition
La plus forte post-condition est générée en partant de la pré-condition et en procédant vers l'avant depuis la première instruction du programme. Chaque affectation introduit un quantificateur existentiel pour représenter la valeur précédente de la variable modifiée.

$\texttt{sp TresSimple}\ (\texttt{x} = 0)\ =$
$\quad \texttt{sp}(\texttt{x} := \texttt{x} + 1; \texttt{x} := \texttt{x} + 2;)\quad (\texttt{x} = 0)\ =$
$\quad \texttt{sp}(\texttt{x} := \texttt{x} + 2;)\quad (\exists \texttt{v}_1.\ \texttt{x} = \texttt{v}_1 + 1 \wedge \texttt{v}_1 = 0) =$
$\quad (\exists v_2.\ x = v_2 + 2 \wedge (\exists v_1.\ v_2 = v_1 + 1 \wedge v_1 = 0))$

Pour montrer le triplet de Hoare {x=0}x :=x+1 ;x :=x+2 ;{x=3} il faut alors montrer que $\texttt{sp TresSimple}\ (\texttt{x} = 0) \Rightarrow \texttt{x=3}$ c'est-à-dire que $(\exists v_2.\ x = v_2 + 2 \wedge (\exists v_1.\ v_2 = v_1 + 1 \wedge v_1 = 0)) \Rightarrow\ \texttt{x=3}$ ce qui est vérifié.
Forme SSA
La forme SSA du programme est calculée en introduisant un renommage de la variable à chaque nouvelle définition (voir figure III.3 section III.2.2 page 97).
SSA(x :=x+1 ;x :=x+2 ;)=
$\quad x_1 = x_0 + 1; x_2 = x_1 + 2$
Pour montrer le triplet de Hoare, il faut alors montrer que la conjonction de la pré-condition (où les variables ont leur renommage initial, x_0 ici) et du programme en forme SSA implique la post-condition (où les variables ont leur renommage final, x_2 ici). Il faut donc montrer que $(x_0 = 0 \wedge (x_1 = x_0 + 1 \wedge x_2 = x_1 + 2)) \Rightarrow x_2 = 3$ ce qui est aussi trivialement vérifié. Notons que cette formule est exactement celle obtenue pour la plus forte post-condition si l'on skolémise l'expression en prenant la constante x_0 pour v_1, la constante x_1 pour v_2 et que l'on renomme x en x_2. ♯

IV.3.2 Exécution symbolique et preuve en avant

Le calcul naïf de $\texttt{sp}\,SP$ par les équations de la figure IV.8 produit des formules compliquées avec des quantifications existentielles imbriquées. Cependant, une stratégie plus fine est possible et constitue un cadre théorique pour l'exécution symbolique en vérification de logiciels [12].

Supposons que toutes les variables d'un programme S soient incluses dans la liste $X_1,\ldots,X_n$ (où quand $m \neq n$ alors $X_m \neq X_n$). On peut alors spécifier un ensemble

$$
\begin{array}{ll}
\mathtt{sp}\,\mathtt{SKIP}\,P & = P \\
\mathtt{wp}\,\mathtt{SKIP}\,Q & = Q \\
\\
\mathtt{sp}\,(V\,\mathtt{:=}\,E)\,P & = \exists v.\ (V{=}E[v/V]) \wedge P[v/V] \\
\mathtt{wp}\,(V\,\mathtt{:=}\,E)\,Q & = Q[E/V] \\
\\
\mathtt{sp}\,(S_1\,;S_2)\,P & = \mathtt{sp}\,S_2\,(\mathtt{sp}\,S_1\,P) \\
\mathtt{wp}\,(S_1\,;S_2)\,Q & = \mathtt{wp}\,S_1\,(\mathtt{wp}\,S_2\,Q) \\
\\
\mathtt{sp}\,(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2)\,P & = (\mathtt{sp}\,S_1\,(P \wedge B)) \ \vee\ (\mathtt{sp}\,S_2\,(P \wedge \neg B)) \\
\mathtt{wp}\,(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2)\,Q & = ((\mathtt{wp}\,S_1\,Q) \wedge B) \ \vee\ ((\mathtt{wp}\,S_2\,Q) \wedge \neg B) \\
\\
\mathtt{sp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,P & = (\mathtt{sp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,(\mathtt{sp}\,S\,(P \wedge B))) \ \vee\ (P \wedge \neg B) \\
\mathtt{wp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,Q & = (\mathtt{wp}\,S\,(\mathtt{wp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,Q) \wedge B) \ \vee\ (Q \wedge \neg B)
\end{array}
$$

FIG. IV.8 – Equations définissant les plus fortes post-conditions (`sp`) et plus faibles pré-conditions (`wp`)

d'états des variables du programme symboliquement par des formules logiques de la forme :

$\exists x_1 \cdots x_n.\ X_1 = e_1 \ \wedge\ \cdots\ \wedge\ X_n = e_n \ \wedge\ \phi$

où x_1,...,x_n sont des variables logiques (i.e. x_i représente symboliquement la valeur initiale de la variable X_i), e_1, ...,e_n sont des expressions (e_i représente la valeur courante de X_i) et ϕ est une formule logique qui établit des relations entre les valeurs des variables (i.e. post-condition en cours de construction pour le calcul de la plus forte post-condition par exécution symbolique). Pour éviter les captures de variables lors des substitutions, e_1,...,e_n et ϕ ne doivent pas contenir les variables du programme X_1,...,X_n ; par contre, elles peuvent très bien contenir les variables logiques x_1,...,x_n. Par exemple, la formule

$\exists i\ j.\ \mathtt{I} = i \ \wedge\ \mathtt{J} = j \ \wedge\ i < j$

représente l'ensemble des états dans lesquels la valeur de la variable du programme `I` (représentée symboliquement par i) est plus petite que la valeur de la variable du programme `J` (représentée symboliquement par j). Cette formule est logiquement équivalente à $\mathtt{I} < \mathtt{J}$.

En général, chaque prédicat P peut se réécrire :

$\exists x_1 \cdots x_n.\ X_1 = x_1 \ \wedge\ \cdots\ \wedge\ X_n = x_n \ \wedge\ P[x_1, \ldots, x_n/X_1, \ldots, X_n]$

où $P[x_1, \ldots, x_n/X_1, \ldots, X_n]$ (qui correspond au ϕ précédent) dénote le résultat du remplacement des occurrences de la variable du programme X_i par la variable x_i ($1 \leq i \leq n$) qui représente symboliquement sa valeur.

Ainsi, calculer `sp` $(X_i\,\mathtt{:=}\,E)$ consiste à évaluer E dans l'état courant (i.e. $E[e_1, \ldots, e_n/X_1, \ldots, X_n]$) et mettre à jour l'équation pour X_i pour spécifier que c'est la nouvelle valeur après l'exécution symbolique de l'affectation.

Si $X_1, \ldots, X_n$, $x_1, \ldots, x_n$ et $e_1, \ldots, e_n$ sont libres par rapport au contexte, alors on peut les abréger en $\overline{X}$, $\overline{x}$ et $\overline{e}$ respectivement. Nous pouvons aussi écrire $\overline{X} = \overline{e}$ pour $X_1 = e_1 \wedge \ldots \wedge X_n = e_n$. Avec ces notations, la règle d'affectation devient :

$\mathtt{sp}\,(X_i\,\mathtt{:=}\,E)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$

$= \exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_i = E[\overline{e}/\overline{X}] \wedge \ldots \wedge X_n = e_n \wedge \phi$

Puisque $\mathtt{sp}\,(S_1\,;S_2)\,P \ =\ \mathtt{sp}\,S_2\,(\mathtt{sp}\,S_1\,P)$, si S_1 et S_2 sont des affectations et que

P a la forme $\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi$, alors pour calculer $\mathtt{sp}\,(S_1\,;S_2)\,P$ il suffit de mettre à jour les équations dans la conjonction correspondant à la variable affectée par S_1 suivi de celle affectée par S_2.

L'exemple IV.6 reprend le calcul de la plus forte post-condition du programme *TresSimple* de l'exemple IV.5 en montrant comment sont effectuées les affectations pour l'exécution symbolique.

Exemple IV.6 (Plus forte post-condition par exécution symbolique) Soit le triplet de Hoare {x=0}x :=x+1 ;x :=x+2 ;{x=3} de l'exemple IV.5. Alors la plus forte post-condition est calculée de la façon suivante avec la règle d'affectation pour l'exécution symbolique :

$\mathtt{sp\ TresSimple}\ (\mathtt{X} = 0)\ =$

$\mathtt{sp}(\mathtt{X} := \mathtt{X} + 1; \mathtt{X} := \mathtt{X} + 2;)\quad (\exists \mathtt{x}\ \mathtt{X} = \mathtt{x} \wedge \mathtt{x} = 0)\ =$

(*par introduction de* $\exists$ *dans* $x = 0$)

$\mathtt{sp}(\mathtt{x} := \mathtt{X} + 2;)\quad (\exists \mathtt{x}\ \mathtt{X} = \mathtt{X{+}1}[\mathtt{x}/\mathtt{X}] \wedge \mathtt{x} = 0)\ =$

(*par la règle d'affectation*)

$\mathtt{sp}(\mathtt{x} := \mathtt{X} + 2;)\quad (\exists \mathtt{x}\ \mathtt{X} = \mathtt{x} + 1 \wedge \mathtt{x} = 0)\ =$

(*après substitution de* X *par* x)

$(\exists x\ \mathtt{X} = \mathtt{X{+}2}[(x+1)/X] \wedge x = 0)\ =$

(*par la règle d'affectation*)

$(\exists x\ \mathtt{X} = (x+1) + 2 \wedge x = 0)$

(*après substitution de* X *par* $x + 1$)

Dans cette dérivation, la fonction ϕ ci-dessus est le prédicat $x = 0$. Notons que l'intérêt majeur de cette nouvelle approche par rapport aux équations classiques de sp est qu'elle ne génère qu'un seul quantificateur existentiel par variable du programme, pour représenter sa valeur initiale. ♯

Pour les conditionnelles, l'équation pour calculer la plus forte post-condition est :

$\mathtt{sp}\,(\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2)\,P\ =\ (\mathtt{sp}\,S_1\,(P \wedge B))\ \vee\ (\mathtt{sp}\,S_2\,(P \wedge \neg B))$.

Si P a la forme $\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi$ alors $P \wedge B$ et $P \wedge \neg B$ peuvent également être mis sous cette même forme. Par conséquent, si une conditionnelle est dans une séquence alors comme $\mathtt{sp}\,S\,(P_1 \vee P_2) = \mathtt{sp}\,S\,P_1 \vee \mathtt{sp}\,S\,P_2$ pour tout programme S, il suit que :

$\mathtt{sp}\,((\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2)\,;S_3)\,P\ =$

$\mathtt{sp}\,(S_1\,;S_3)\,(P \wedge B)\ \vee\ \mathtt{sp}\,(S_2\,;S_3)\,(P \wedge \neg B)$

Cela établit que le calcul de la plus forte post-condition d'une séquence qui commence par une conditionnelle peut être effectué par des exécutions symboliques séparées. Donc si l'on peut montrer que $P \wedge B$ ou $P \wedge \neg B$ sont faux alors, puisque pour chaque S, $\mathtt{sp}\,S\,\mathsf{F} = \mathsf{F}$, une des disjonctions peut être supprimée. Sinon, l'exécution symbolique des deux branches doit être faite (en appliquant différentes heuristiques d'exploration en profondeur ou en largeur d'abord).

L'exemple IV.7 illustre l'exécution symbolique d'instructions conditionnelles. Il montre la génération de la plus forte post-condition du programme *AbsMinus* présenté dans la sous-section IV.2 figure IV.3, quand la pré-condition est $i < j$. C'est la formalisation exacte de l'exécution symbolique détaillée dans la sous-section IV.2, en particulier en ce qui concerne le calcul de l'état. En effet, $\exists \overline{x}.\ \overline{X} = \overline{e}$ formalise

la gestion de l'état par des listes associatives où X_i sont les symboles des associations, x_i sont les valeurs initiales associées et e_i les valeurs courantes associées. De plus, dans les formules ci-dessus, ϕ est la post-condition en cours de construction : il s'agit de la pré-condition à l'état initial et de la conjonction de la pré-condition et des décisions prises sur le chemin courant pour les états de calcul intermédiaires.

Exemple IV.7 (Plus forte post-condition pour le programme $AbsMinus$**)** Pour chaque transformation, le terme qui a été introduit ou modifié est souligné.

$\mathtt{sp\ AbsMinus}\ (\mathtt{I} < \mathtt{J})\ =$

$\mathtt{sp}(\mathtt{R} := 0;$
$\quad \mathtt{K} := 0;$
$\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP};$
$\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$(\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = j \wedge \mathtt{K} = k \wedge \mathtt{R} = r \wedge i < j)\ =$

$\mathtt{sp}(\mathtt{K} := 0;$
$\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP};$
$\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$(\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = k \wedge \underline{\mathtt{R} = 0} \wedge i < j)\ =$

$\mathtt{sp}(\mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP};$
$\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$(\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \underline{\mathtt{K} = 0} \wedge \mathtt{R} = 0 \wedge i < j)\ =$

$(\mathtt{sp}(\mathtt{K} := \mathtt{K} + 1;\ \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \underline{(\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}]}))$
$\vee$
$\mathtt{sp}(\mathtt{SKIP};\ \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \underline{\neg(\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}]})))$

Puisque $(\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}] = i < j$ la pré-condition de la seconde disjonction contient la conjonction $i < j \wedge \neg(i < j)$ qui est fausse. Donc la seconde disjonction peut être supprimée pour obtenir :

$\mathtt{sp}(\mathtt{K} := \mathtt{K} + 1;\ \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge i < j))\ =$

$\mathtt{sp}(\mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \underline{\mathtt{K} = (\mathtt{K{+}1})[0/\mathtt{K}]} \wedge \mathtt{R} = 0 \wedge i < j))\ =$

$(\mathtt{sp}(\mathtt{R} := \mathtt{J} - \mathtt{I})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 1 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \underline{(1 = 1 \wedge \neg(i = j))}))$
$\vee$
$\mathtt{sp}(\mathtt{R} := \mathtt{I} - \mathtt{J})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \underline{\neg(1 = 1 \wedge \neg(i = j))})))\ =$
$\quad \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 1 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \neg(1 = 1 \wedge \neg(i = j))))\ =$

La seconde disjonction est supprimée car $i < j \wedge \neg(1 = 1 \wedge \neg(i = j))$ se simplifie en F.

$\mathtt{sp}(\mathtt{R} := \mathtt{J} - \mathtt{I})$
$\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 1 \wedge \mathtt{R} = 0 \wedge i < j)\ =$
$(\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 1 \wedge \underline{\mathtt{R} = (\mathtt{J}{-}\mathtt{I})[i, j/\mathtt{I}, \mathtt{J}]} \wedge i < j)$

La partie droite de cette équation se simplifie en $\mathtt{R} = \mathtt{J}-\mathtt{I} \wedge \mathtt{I} < \mathtt{J}$ en effectuant la substitution et en utilisant les propriétés des quantificateurs existentiels. Finalement : $\mathtt{sp\ AbsMinus}\ (\mathtt{I} < \mathtt{J}) = \mathtt{R} = \mathtt{J}-\mathtt{I} \wedge \mathtt{I} < \mathtt{J}$. ♯

IV.3.3 Boucles et invariants

Le BMC déplie les boucles jusqu'à une certaine profondeur. La vérification est donc plutôt une recherche d'erreurs : en cas de succès, on peut seulement affirmer que le programme ne contient pas d'erreurs jusqu'à cette profondeur. D'autre part, il n'y a pas de façon générale pour calculer la plus forte post-condition ou la plus faible pré-condition pour les boucles `WHILE` : la réécriture avec les équations de la figure IV.8 peut ne pas terminer. Une façon d'effectuer une preuve formelle et complète est donc d'utiliser des invariants de boucle qui peuvent être fournis par un humain ou par un algorithme de génération d'invariants. La logique de Hoare fournit la règle `WHILE` suivante :

$$\frac{\vdash P \Rightarrow R \qquad \vdash \{R \wedge B\}\, S\, \{R\} \qquad R \wedge \neg B \Rightarrow Q}{\vdash \{P\}\, \mathtt{WHILE}\, B\, \mathtt{DO}\{R\}\, S\, \{Q\}}$$

où $\{R\}$ est une assertion qui représente l'invariant. Cette règle est la base logique de toutes les méthodes qui utilisent un invariant pour vérifier des programmes contenant des boucles. Elle stipule que le triplet $\{\mathrm{P}\}\,\mathtt{WHILE}\,\mathrm{B}\,\mathtt{DO}\{\mathrm{R}\}\,\mathrm{S}\,\{\mathrm{Q}\}$ est correct si l'invariant est impliqué par la pré-condition ($P \Rightarrow R$), si l'invariant est préservé par exécution d'une étape de la boucle $\{R \wedge B\} S \{R\}$ et que la conjonction de l'invariant et de la condition de sortie de boucle implique Q ($R \wedge \neg B \Rightarrow Q$).

La génération des plus fortes post-conditions de programmes avec boucle en utilisant la règle `WHILE` ci-dessus, est distinguée de la génération classique (i.e. sans utiliser d'invariant) en l'appelant "*approximation* de la plus forte post-condition" (en ce sens que l'invariant peut donner un résultat approximé de l'exécution de la boucle). Les règles $\mathtt{asp}\, S\, P$ pour calculer l'approximation de la plus forte post-condition sont les mêmes que celles pour calculer $\mathtt{sp}\, S\, P$ sauf que la règle $\mathtt{asp}\, \mathtt{WHILE}\, B\, \mathtt{DO}\{R\}\, S\, P$ a été ajoutée : $\mathtt{asp}\,(\mathtt{WHILE}\, B\, \mathtt{DO}\{R\}\, S)\, P = R \wedge \neg B$

En effet, c'est la partie de la règle `WHILE` qui concerne la vérification de la post-condition. Dans le cadre de l'exécution symbolique, cette équation peut être ré-écrite en :

$$\begin{aligned}&\mathtt{asp}\,(\mathtt{WHILE}\, B\, \mathtt{DO}\{R\}\, S)\,(\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi)\\ &= \exists \overline{x}.\ \overline{X} = \overline{x} \wedge (R \wedge \neg B)[\overline{x}/\overline{X}]\end{aligned}$$

Ainsi, exécuter symboliquement $\mathtt{WHILE}\, B\, \mathtt{DO}\{R\}\, S$ consiste à supprimer la pré-condition et à recommencer l'exécution dans un nouvel état symbolique qui correspond à l'état spécifié comme vrai après la boucle par la règle `WHILE` de Hoare.

Pour effectuer correctement la vérification d'un triplet de Hoare en utilisant la règle `WHILE`, il faut s'assurer que le contexte de son utilisation est correct c'est-à-dire que la pré-condition implique l'invariant et que l'invariant en est bien un. Les méthodes de preuve basées sur ce principe génèrent donc deux types d'information : d'une part l'approximation de la plus forte post-condition (ou plus faible pré-condition), et d'autre part un ensemble de *conditions de vérification* notées $\mathtt{svc}\,\mathrm{S}\,\mathrm{P}$

dans le cas en avant pour "Strongest Verification Conditions"[12]. Ces conditions sont des assertions qui doivent être vérifiées pour assurer que les règles de génération de l'approximation de la plus forte post-condition soient appliquées correctement. En particulier, pour le `WHILE`, il faut que :

$(P \Rightarrow R) \wedge (\mathtt{asp}\, S\, (R \wedge B) \Rightarrow R) \wedge \mathtt{svc}\, S\, (R \wedge B)$

où $\mathtt{svc}\, S\, (R \wedge B)$ est la condition de vérification associée à S pour la pré-condition $R \wedge B$.

Dans le cadre de l'exécution symbolique il faut donc que :

$$\begin{aligned}&((\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi) \Rightarrow R)\\ &\quad \wedge (\mathtt{asp}\, S\, (R \wedge B) \Rightarrow R) \wedge \mathtt{svc}\, S\, (R \wedge B)\\ &= (\forall \overline{x}.\ \phi \Rightarrow R[\overline{e}/\overline{X}]) \wedge (\mathtt{asp}\, S\, (R \wedge B) \Rightarrow R) \wedge \mathtt{svc}\, S\, (R \wedge B)\end{aligned}$$

Cela signifie que la pré-condition doit impliquer l'invariant évalué dans l'état avant exécution de la boucle ($\forall \overline{x}.\ \phi \Rightarrow R[\overline{e}/\overline{X}]$), que l'invariant R doit en être un ($\mathtt{asp}\, S\, (R \wedge B) \Rightarrow R$) et que les conditions de vérification de S sont satisfaites à chaque itération ($\mathtt{svc}\, S\, (R \wedge B)$).

IV.4 Conclusion

J'effectue dans cette conclusion une critique des travaux présentés dans ce chapitre, qui sont des travaux récents. Les perspectives de la section VI.2 décrivent les directions futures que je souhaite leur donner.

Les travaux de la section IV.2 m'ont permis d'appréhender la vérification avec un assistant de preuves interactif. Nous avons tenté d'atteindre les objectifs suivants :

1. Effectuer du bounded model checking (BMC) alors que l'utilisation d'un assistant de preuves va usuellement de pair avec une preuve complète,
2. Assurer une plus grande correction à la vérification par BMC que celle offerte par les outils automatiques, d'une part en basant le calcul de l'état sur la sémantique du langage et d'autre part en appliquant des règles de simplification formelles directement avec le prouveur,
3. Rester suffisamment efficace pour être, sinon compétitif, mais tout au moins acceptable, par rapport aux outils purement automatiques comme CBMC (voir section III.4.5).

Le point 1 me semble une voie à poursuivre, mais de façon conjointe à une preuve complète. En effet, les outils de vérification formelle basés sur la génération de plus faible pré-condition (e.g. *Why* présenté section III.4.5 ou *Jack* [3]) échouent en l'absence d'invariants de boucle. Il est donc important d'offrir l'alternative d'une approche par BMC lorsque les invariants ne sont pas fournis, mais il est essentiel d'effectuer ce BMC de façon complémentaire à la preuve complète, dans le même cadre sémantique.

[12]Le lecteur est renvoyé à l'article support IV.5.1 où l'ensemble des règles approximées et des conditions de vérification sont présentées figure 5.

Le point 2 est un objectif difficile à atteindre, et surtout difficile à quantifier. En effet, pour assurer une correction maximale, l'algorithme de BMC tout entier doit être écrit en HOL et exécuté en HOL. Mike Gordon a formalisé l'algorithme de BMC dans HOL4, par un ensemble de définitions et de théorèmes qui établissent la correction de l'algorithme. Cependant, cette version HOL4 est très lente et n'a pas pu être appliquée à des exemples de taille significative (e.g exemples avec des tableaux). D'un autre côté, la version que j'ai implémentée en ML est plus efficace mais n'assure pas le même degré de correction puisqu'une partie du calcul seulement dérive directement des théorèmes HOL4 (i.e. calcul de l'état et règles de simplification). Par contre, l'algorithme en ML n'est pas lui-même formellement vérifié. Le gain en correction est donc difficile à quantifier.

Enfin le point 3 est également difficile à assurer. Notre implémentation a démontré que l'on pouvait effectuer une preuve complètement automatique et dans des temps raisonnables pour un ensemble d'exemples. Cependant, cela ne passe pas à l'échelle pour des programmes "réels" comme le programme "FlasherManager" fourni par un de nos partenaires industriels (voir section VI.1) ou même tout simplement si l'on cherche à traiter des tableaux de grande taille.

La synthèse de la sous-section IV.3 pose la question de l'efficacité des preuves "en avant" par rapport à celle des preuves "en arrière". Le fait de pouvoir couper des branches entières du programme dans l'approche "en avant" laisse espérer une plus grande efficacité, en tout cas pour les programmes où la pré-condition impose de fortes contraintes sur l'état. Cependant, dans l'approche "en arrière", les formules peuvent aussi se simplifier quand la condition courante est incompatible avec la pré-condition en cours de construction, c'est-à-dire en particulier avec la post-condition initiale. Dans ce cas, la formule est simplifiée si une des deux branches ne peut pas aboutir à un état qui satisfait la post-condition. Faut-il alors combiner ces deux approches ? D'autre part, la règle pour l'exécution symbolique du `WHILE` avec invariant est très simple, et pourrait être intégrée à un outil de BMC dans le cas où les invariants sont fournis.

L'utilisation d'un solveur SMT soulève aussi une question majeure sur la pertinence de l'exécution symbolique. En effet, de tels solveurs peuvent traiter de façon très efficace des formules qui contiennent des expressions arithmétiques linéaires, imbriquées dans une structure conditionnelle complexe. Ainsi, les outils de BMC qui reposent sur des solveurs SMT génèrent une énorme forme conditionnelle. Ceci s'oppose à l'exécution symbolique où le choix est de tester pas à pas chaque condition. Dans mes travaux avec HOL4, j'ai pu utiliser une autre version de la sémantique où la fonction de sémantique de la conditionnelle fournit un terme conditionnel qui n'est pas réduit. Je m'en suis servie pour générer directement la formule correspondante dans la syntaxe d'entrée de *Yices*. Les premières expérimentations ont montré que ceci est beaucoup plus efficace que la version avec exécution symbolique. Toutefois, l'exploration des branches dans l'exécution symbolique peut être optimisée, nous reviendrons là-dessus section VI.1.

IV.5 Article support

IV.5.1 Forward with Hoare

M.J.C Gordon, H.Collavizza. Forward with Hoare. *Reflections on the Work of C.A.R. Hoare.* Accepted for publication, to appear in History of Computing Series, Springer.

Forward with Hoare

[Draft dated November 19, 2009]

Mike Gordon[1] and Hélène Collavizza[2]

[1] University of Cambridge Computer Laboratory William Gates Building, 15 JJ Thomson Avenue Cambridge CB3 0FD, UK. Mike.Gordon@cl.cam.ac.uk
[2] Université de Nice–Sophia-Antipolis – I3S/CNRS, 930, route des Colles B.P. 145 06903 Sophia-Antipolis, France. helen@polytech.unice.fr

Summary. Hoare's celebrated paper entitled "An Axiomatic Basis for Computer Programming" appeared in 1969, so the Hoare formula $P\{S\}Q$ is now forty years old! That paper introduced Hoare Logic, which is still the basis for program verification today, but is now mechanised inside sophisticated verification systems. We aim here to give an accessible introduction to methods for proving Hoare formulae based both on the forward computation of postconditions and on the backwards computation of preconditions. Although precondition methods are better known, computing postconditions provides a verification framework that encompasses methods ranging from symbolic execution to full deductive proof of correctness.

1 Introduction

Hoare logic [12] is a deductive system whose axioms and rules of inference provide a method of proving statements of the form $P\{S\}Q$, where S is a program statement[3] and P and Q are assertions about the values of variables. Following current practise, we use the notation $\{P\}\,S\,\{Q\}$ instead of $P\{S\}Q$. Such a "Hoare triple" means that Q (the "postcondition") holds in any state reached by executing S from an initial state in which P (the "precondition") holds. Program statements may contain variables V (`X`, `Y`, `Z` etc.), value expressions (E) and Boolean expressions (B). They are built out of the skip (`SKIP`) and assignment statements (V `:=` E) using sequential composition (S_1`;`S_2), conditional branching (`IF` B `THEN` S_1 `ELSE` S_2) and `WHILE`-loops (`WHILE` B `DO` S). The assertions P and Q are formal logic formulae expressing properties of the values of variables.

Hoare explicitly acknowledges that his deductive system is influenced by the formal treatment of program execution due to Floyd [8].[4] There is, how-

[3] The word "statement" is overused: Hoare statements $P\{S\}Q$ (or $\{P\}S\{Q\}$) are either true or false, but program statements are constructs that can be executed to change the values of variables. To avoid this confusion program statements are sometimes called commands.

[4] The fascinating story of the flow of ideas between the early pioneers of programming logic is delightfully told in Jones' historical paper [16].

ever, a difference, which is exhibited below using Hoare triple notation (where the notation $M[E/V]$, where M can be a formula or expression, denotes the result of substituting E for V in M).

Floyd's assignment axiom: $\vdash \{P\}\, V \texttt{:=} E\, \{\exists v.\ (V = E[v/V]) \wedge P[v/V]\}$

Hoare's assignment axiom: $\vdash \{Q[E/V]\}\, V \texttt{:=} E\, \{Q\}$

These are axiom schemes: any instance obtained by replacing P, Q, V, E by specific terms and formulae is an axiom. Example instances of the axiom schemes, using the replacements $P \mapsto (\texttt{X=Y})$, $Q \mapsto (\texttt{X=2}\times\texttt{Y})$, $V \mapsto \texttt{X}$ and $E \mapsto (\texttt{X+Y})$, are:

Floyd: $\vdash \{\texttt{X=Y}\}\, \texttt{X := X+Y}\, \{\exists v.\ (\texttt{X} = ((\texttt{X+Y})[v/\texttt{X}])) \wedge ((\texttt{X=Y})[v/\texttt{X}])\}$

Hoare: $\vdash \{(\texttt{X=2}\times\texttt{Y})[(\texttt{X+Y})/\texttt{X}]\}\, \texttt{X := X+Y}\, \{\texttt{X=2}\times\texttt{Y}\}$

which become the following if the substitutions $M[E/V]$ are performed:

Floyd: $\vdash \{\texttt{X=Y}\}\, \texttt{X := X+Y}\, \{\exists v.\ (\texttt{X} = v\texttt{+Y}) \wedge (v\texttt{=Y})\}$

Hoare: $\vdash \{(\texttt{X+Y})\texttt{=2}\times\texttt{Y}\}\, \texttt{X := X+Y}\, \{\texttt{X=2}\times\texttt{Y}\}$

These are both equivalent to $\vdash \{\texttt{X=Y}\}\, \texttt{X := X+Y}\, \{\texttt{X=2}\times\texttt{Y}\}$, but the reasoning in the Hoare case is a bit simpler since there is no existential quantification.

In general, the Floyd and Hoare assignment axioms are equivalent, but it is the Hoare axiom that is more widely used, since it avoids an accumulation of an existential quantifier – one for each assignment in the program.

The axioms of Hoare logic include all instances of the Hoare assignment axiom scheme given above. The rules of inference of the logic provide rules for combining Hoare triples about program statements into Hoare triples about the result of combining the statements using sequential composition, conditional branches and `WHILE`-loops.

2 Weakest preconditions and strongest postconditions

A few years after Hoare's pioneering paper, Dijkstra published his influential book "A Discipline of Programming" [6] in which a framework for specifying semantics based on 'predicate transformers' – rules for transforming predicates on states – is described. Dijkstra regarded assertions like preconditions (P above) and postconditions (Q above) as predicates on the program state, since for a given state such an assertion is either true or false. His book introduces 'weakest preconditions' as a predicate transformer semantics that treats assignment statements in a way equivalent to Hoare's assignment axiom. A dual notion of 'strongest postconditions' corresponds to Floyd's treatment of assignments. We do not know who first introduced the concept of strongest postconditions. They are discussed in Dijkstra's 1990 book with Scholten [7], but several recent papers (e.g. [11]) cite Gries' 1981 textbook [10], which only describes them in an exercise. Jones [16, p. 12] mentions that Floyd discusses a clearly related notion of 'strongest verifiable consequents' in his 1967 paper.

If $P \Rightarrow Q$ then P is said to be stronger than Q. The strongest postcondition predicate transformer for a statement S transforms a precondition predicate P to a postcondition predicate $\mathtt{sp}\,S\,P$, which is the 'strongest' predicate holding after executing S in a state satisfying precondition P. This is strongest in the sense that if Q has the property that it holds of any state resulting from executing S when P, then $\mathtt{sp}\,S\,P$ is stronger than Q – i.e. $\mathtt{sp}\,S\,P \Rightarrow Q$. The strongest postcondition predicate transformer semantics of $V := E$ is:

$$\mathtt{sp}\,(V := E)\,P \;=\; \exists v.\ (V = E[v/V]) \wedge P[v/V]$$

The definition of strongest postcondition entails that $\{P\}\,V := E\,\{Q\}$ holds if and only if $\mathtt{sp}\,(V := E)\,P$ entails Q.

If $P \Rightarrow Q$ then Q is said to be weaker than P. The weakest precondition predicate transformer for a statement S transforms a postcondition predicate Q to a precondition predicate $\mathtt{wp}\,S\,Q$, which is the 'weakest' predicate that ensures that if a state satisfies it then after executing S the predicate Q holds.[5] This is weakest in the sense that if P has the property that executing S when P holds ensures that Q holds, then $\mathtt{wp}\,S\,Q$ is weaker than P – i.e. $P \Rightarrow \mathtt{wp}\,S\,Q$. The weakest precondition predicate transformer semantics of $V := E$ is:

$$\mathtt{wp}\,(V := E)\,Q \;=\; Q[E/V]$$

The definition of weakest precondition entails that $\{P\}\,V := E\,\{Q\}$ holds if and only if P entails $\mathtt{wp}\,(V := E)\,Q$.

Equations satisfied by $\mathtt{sp}\,S\,P$ and $\mathtt{wp}\,S\,Q$ are listed in Fig. 1 (the equations for assignments are repeated there for convenience). These equations were originally taken as axioms. The axiomatic approach is discussed in Hoare's paper: it has the advantage of allowing a partial specification of meaning, which gives freedom to compiler writers and can make language standards more flexible. However, a purely axiomatic approach is hard to scale to complex programming constructs, as the axioms and rules get complicated and consequently hard to trust. It is now more common to give a formal semantics of programming constructs (either operational or denotational) and to derive Hoare logic axioms and rules and predicate transformer laws from this [24].

Computing $\mathtt{sp}\,(S_1\,;S_2)\,P$ using the equations in Fig. 1 consists of starting from a precondition P, then first computing $\mathtt{sp}\,S_1\,P$ and then applying $\mathtt{sp}\,S_2$ to the resulting predicate. This is forwards symbolic execution. In contrast, computing $\mathtt{wp}\,(S_1\,;S_2)\,Q$ proceeds backwards from a postcondition Q by first computing $\mathtt{wp}\,S_2\,Q$ and then applying $\mathtt{wp}\ S_1$ to the resulting predicate. We have more to say about forwards versus backwards later (e.g. in Section 6).

[5] Dijkstra defined "weakest precondition" to require termination of S – what we are calling "weakest precondition" he calls "weakest liberal precondition". Dijkstra also uses different notation: in his first book he uses `wlp(`S`,`Q`)` and `wp(`S`,`Q`)`. In the later book with Scholten he uses $\mathtt{wlp}.S.Q$ and $\mathtt{wp}.S.Q$. Thus our $\mathtt{wp}\,S\,Q$ is Dijkstra's `wlp(`S`,`Q`)` (or $\mathtt{wlp}.S.Q$). However, our use of "strongest postcondition" corresponds to Dijkstra's, though our notation differs.

$\mathtt{sp}\,\mathtt{SKIP}\,P = P$
$\mathtt{wp}\,\mathtt{SKIP}\,Q = Q$

$\mathtt{sp}\,(V\,\mathtt{:=}\,E)\,P = \exists v.\ (V{=}E[v/V]) \wedge P[v/V]$
$\mathtt{wp}\,(V\,\mathtt{:=}\,E)\,Q = Q[E/V]$

$\mathtt{sp}\,(S_1\,;S_2)\,P = \mathtt{sp}\,S_2\,(\mathtt{sp}\,S_1\,P)$
$\mathtt{wp}\,(S_1\,;S_2)\,Q = \mathtt{wp}\,S_1\,(\mathtt{wp}\,S_2\,Q)$

$\mathtt{sp}\,(\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2)\,P = (\mathtt{sp}\,S_1\,(P \wedge B)) \ \vee\ (\mathtt{sp}\,S_2\,(P \wedge \neg B))$
$\mathtt{wp}\,(\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2)\,Q = ((\mathtt{wp}\,S_1\,Q) \wedge B) \ \vee\ ((\mathtt{wp}\,S_2\,Q) \wedge \neg B)$

$\mathtt{sp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,P = (\mathtt{sp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,(\mathtt{sp}\,S\,(P \wedge B))) \ \vee\ (P \wedge \neg B)$
$\mathtt{wp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,Q = (\mathtt{wp}\,S\,(\mathtt{wp}\,(\mathtt{WHILE}\,B\,\mathtt{DO}\,S)\,Q) \wedge B) \ \vee\ (Q \wedge \neg B)$

Fig. 1. Equations defining strongest postconditions and weakest preconditions

3 Proving Hoare triples via predicate transformers

The relationship between Hoare triples, strongest postconditions and weakest preconditions is that $\{P\}S\{Q\}$ holds if and only if $(\mathtt{sp}\,S\,P) \Rightarrow Q$ and also if and only if $P \Rightarrow \mathtt{wp}\,S\,Q$. These implications are purely logical formulae, so a pure logic theorem prover can be used to prove them. Thus strongest postconditions and weakest preconditions each provide a way of 'compiling' the problem of verifying a Hoare triple to a purely logical problem.[6] For a loop-free program S, the equations in Fig. 1 can be used as left-to-right rewrites to calculate $\mathtt{sp}\,S\,P$ and $\mathtt{wp}\,S\,Q$ (if S contains a `WHILE`-loop then such rewriting may not terminate). The right hand side of the equation for $\mathtt{sp}\,(V\,\mathtt{:=}\,E)\,P$ contains an existentially quantified conjunction, whereas the right hand side of the equation for $\mathtt{wp}\,(V\,\mathtt{:=}\,E)\,Q$ is just $Q[E/V]$, thus the formulae generated by the equations for strongest postconditions will be significantly more complex than those for weakest preconditions. This is one reason why weakest preconditions are often used in Hoare logic verifiers. The reader is invited to compare the two proofs of $\{\mathtt{I} < \mathtt{J}\}\,\mathtt{AbsMinus}\,\{\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J}\}$ (the loop-free program `AbsMinus` is given in Fig. 2) obtained by manually calculating $\mathtt{sp}\,\mathtt{AbsMinus}\,(\mathtt{I} < \mathtt{J})$ and $\mathtt{wp}\,\mathtt{AbsMinus}\,(\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J})$.

Although the naive calculation of $\mathtt{sp}\,S\,P$ using the equations in Fig. 1 generates complicated formulae with nested existential quantifications, a more careful calculation strategy based on symbolic execution is possible. This can be used as a theoretical framework for symbolic execution in software model checking [11].

[6] Actually this is an oversimplification: mathematical constants might occur in the formula, e.g. $+$, $-$, $\times$ from the theory of arithmetic, so the theorem prover may need to go beyond pure logic and solve problems in mathematical theories.

```
R:=0;
K:=0;
IF I < J THEN K:=K+1 ELSE SKIP ;
IF K = 1 ∧ ¬(I = J) THEN R:=J−I ELSE R:=I−J
```

Fig. 2. The loop-free example program `AbsMinus`

4 Symbolic execution and strongest postconditions

Suppose all the variables in a program S are included in the list X_1,...,X_n (where if $m \neq n$ then $X_m \neq X_n$). We shall specify a set of states of the program variables symbolically by logical formulae of the form:

$$\exists x_1 \cdots x_n.\ X_1 = e_1 \ \wedge \ \cdots \ \wedge \ X_n = e_n \ \wedge \ \phi$$

where x_1,...,x_n are logical variables (think of x_i as symbolically representing the initial value of program variable X_i), e_1, ...,e_n are expressions (e_i represents the current value of X_i) and ϕ is a logical formula constraining the relationships between the values of the variables. For reasons that will become clear later, it is required that neither e_1,...,e_n nor ϕ contain the program variables X_1,...,X_n, though they may well contain the variables x_1,...,x_n. For example, the formula

$$\exists i\ j.\ \mathtt{I} = i \ \wedge \ \mathtt{J} = j \ \wedge \ i < j$$

represents the set of states in which the value of program variable `I` (represented symbolically by i) is less than the value of program variable `J` (represented symbolically by j). This formula is logically equivalent to $\mathtt{I} < \mathtt{J}$. In general, any predicate P can be written as:

$$\exists x_1 \cdots x_n.\ X_1 = x_1 \ \wedge \ \cdots \ \wedge \ X_n = x_n \ \wedge \ P[x_1, \ldots, x_n / X_1, \ldots, X_n]$$

where $P[x_1, \ldots, x_n / X_1, \ldots, X_n]$ (corresponding to ϕ above) denotes the result of replacing all occurrences of program variable X_i by variable x_i ($1 \leq i \leq n$) that symbolically represents its value. In these formulae, program variables X_i and variables x_i, representing symbolic values, are both just logical variables. Expressions in programs (e.g. the right hand side of assignments) are logic terms and tests in conditionals are logic formulae. This identification of program language constructs with logic terms and formulae is one of the reasons why Hoare logic is so effective. Although this identification might appear to be confusing the different worlds of programming languages and logical systems, it does have a sound semantic basis [13, 4, 2]. The reason for adopting this form of symbolic representation is because the strongest postcondition for assignment preserves it and introduces no new existential quantifiers.

$$\mathtt{sp}\,(X_i := E)\,(\exists x_1 \cdots x_n.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi) \ =$$
$$\exists x_1 \cdots x_n.\ X_1 = e_1 \wedge \cdots \wedge X_i = E[e_1, \ldots, e_n / X_1, \ldots, X_n] \wedge \cdots \wedge X_n = e_n \wedge \phi$$

Thus calculating $\mathtt{sp}\ (X_i := E)$ consists of evaluating E in the current state (i.e. $E[e_1, \ldots, e_n/X_1, \ldots, X_n]$) and then updating the equation for X_i to specify that this is the new value after the symbolic execution of the assignment.

If $X_1, \ldots, X_n$, $x_1, \ldots, x_n$ and $e_1, \ldots, e_n$ are clear from the context, then they may be abbreviated to $\overline{X}$, $\overline{x}$ and $\overline{e}$ respectively. We may also write $\overline{X} = \overline{e}$ to mean $X_1 = e_1 \wedge \ldots \wedge X_n = e_n$. With this notation the equation above becomes:

$$\begin{array}{l}\mathtt{sp}\,(X_i := E)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) \\ = \exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_i = E[\overline{e}/\overline{X}] \wedge \ldots \wedge X_n = e_n \wedge \phi\end{array}$$

The derivation of this equation follows below (an informal justification of each line is given in brackets just after the line). The validity of the equation depends of the restriction that neither e_1,...,e_n nor ϕ contain the program variables X_1,...,X_n. In addition, we also need to assume below that v, x_1,...,x_n and X_1,...,X_n are all distinct and v, x_1,...,x_n do not occur in E. These restrictions are assumed from now on.

$\mathtt{sp}\,(X_i := E)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$

$= \exists v.\ \mathtt{X}_i{=}E[v/\mathtt{X}_i] \wedge (\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)[v/\mathtt{X}_i]$
(Floyd assignment rule)

$= \exists v.\ \mathtt{X}_i{=}E[v/\mathtt{X}_i] \wedge (\exists \overline{x}.\ \mathtt{X}_1{=}e_1 \wedge \cdots \wedge v{=}e_i \wedge \cdots \wedge \mathtt{X}_n{=}e_n \wedge \phi)$
(distinctness of variables and X_i not in e_1,...,e_n or ϕ)

$= \exists v\ \overline{x}.\ \mathtt{X}_i{=}E[v/\mathtt{X}_i] \wedge \mathtt{X}_1{=}e_1 \wedge \cdots \wedge v{=}e_i \wedge \cdots \wedge \mathtt{X}_n{=}e_n \wedge \phi$
(pulling quantifiers to front: allowed as variables distinct, $\overline{x}$ not in E)

$= \exists \overline{x}.\ \mathtt{X}_i{=}E[e_i/\mathtt{X}_i] \wedge \mathtt{X}_1{=}e_1 \wedge \cdots \wedge (\exists v.\ v{=}e_i) \wedge \cdots \wedge \mathtt{X}_n{=}e_n \wedge \phi$
(restricting scope of v to the only conjunct containing v)

$= \exists \overline{x}.\ \mathtt{X}_i{=}E[e_i/\mathtt{X}_i] \wedge \mathtt{X}_1{=}e_1 \wedge \cdots \wedge \mathsf{T} \wedge \cdots \wedge \mathtt{X}_n{=}\mathtt{e}_n \wedge \phi$
($\exists v.\ v{=}e_i$ is true)

$= \exists \overline{x}.\ \mathtt{X}_1{=}e_1 \wedge \cdots \wedge \mathtt{X}_i{=}E[e_i/\mathtt{X}_i] \wedge \cdots \wedge \mathtt{X}_n{=}e_n \wedge \phi$
(eliminate T and move equation for $\mathtt{X}_i$ to where it was in the conjunction)

$= \exists \overline{x}.\ \mathtt{X}_1{=}e_1 \wedge \cdots \wedge \mathtt{X}_i{=}E[e_1, \ldots, e_n/\mathtt{X}_1, \ldots, \mathtt{X}_n] \wedge \ldots \wedge \mathtt{X}_n{=}e_n \wedge \phi$
($\overline{X} = \overline{e}$ justify replacing $\mathtt{X}_1$,...,$\mathtt{X}_n$ in E by e_1,...,e_n)

$= \exists \overline{x}.\ \mathtt{X}_1{=}e_1 \wedge \cdots \wedge \mathtt{X}_i{=}E[\overline{e}/\overline{\mathtt{X}}] \wedge \cdots \wedge \mathtt{X}_n{=}e_n \wedge \phi$
(definition of $[\overline{e}/\overline{\mathtt{X}}]$ notation)

Since $\mathtt{sp}\,(S_1\,;S_2)\,P\ =\ \mathtt{sp}\,S_2\,(\mathtt{sp}\,S_1\,P)$, if S_1 and S_2 are assignments and P has the form $\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi$, then to compute $\mathtt{sp}\,(S_1\,;S_2)\,P$, one just updates the equations in the conjunction corresponding to the variable being assigned by S_1 followed by that assigned by S_2.

For conditional branches, the equation for calculating strongest postconditions is: $\mathtt{sp}\,(\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2)\,P\ =\ (\mathtt{sp}\,S_1\,(P \wedge B))\ \vee\ (\mathtt{sp}\,S_2\,(P \wedge \neg B))$. If P has the form $\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi$ then $P \wedge B$ and $P \wedge \neg B$ can be put into this form. The derivation is below.

$$
\begin{aligned}
P \wedge B \;&=\; (\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) \wedge B \\
&\quad \text{(expanding } P\text{)} \\
&=\; \exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge B) \\
&\quad \text{(allowed if } x_1,\ldots,x_n \text{ do not occur in } B\text{, which is assumed)} \\
&=\; \exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge B[\overline{e}/\overline{X}]) \\
&\quad \text{(conjuncts } \overline{X} = \overline{e} \text{ justify replacing } \overline{\mathtt{X}} \text{ in } B \text{ by } \overline{e}\text{)}
\end{aligned}
$$

Similarly: $P \wedge \neg B \;=\; \exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge \neg B[\overline{e}/\overline{X}])$.

If a conditional is in a sequence then as $\mathtt{sp}\, S\, (P_1 \vee P_2) = \mathtt{sp}\, S\, P_1 \vee \mathtt{sp}\, S\, P_2$ for any program S, it follows that:

$$
\begin{aligned}
&\mathtt{sp}\,((\mathtt{IF}\, B\, \mathtt{THEN}\, S_1\, \mathtt{ELSE}\, S_2)\,;S_3)\, P \;= \\
&\;\mathtt{sp}\,(S_1\,;S_3)\,(P \wedge B) \;\vee\; \mathtt{sp}\,(S_2\,;S_3)\,(P \wedge \neg B)
\end{aligned}
$$

thus the calculation of the strongest postcondition of a sequence starting with a conditional can proceed by separate symbolic evaluations of each arm.

If it can be shown that either $P \wedge B$ or $P \wedge \neg B$ are false (F) then, since for any S it is the case that $\mathtt{sp}\, S\, \mathsf{F} = \mathsf{F}$, one of the disjuncts can be pruned. If such pruning is not possible, then separate evaluations for each arm must be performed. These can be organised to maximise efficiency based on heuristics (e.g. depth-first or breadth-first). As an example illustrating how symbolic evaluation can be used to compute strongest postconditions, we calculate:

$$
\begin{aligned}
&\mathtt{sp\ AbsMinus}\ (\mathtt{I} < \mathtt{J}) \;= \\
&\mathtt{sp}(\mathtt{R} := 0; \\
&\quad \mathtt{K} := 0; \\
&\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP}; \\
&\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J}) \\
&\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = j \wedge \mathtt{K} = k \wedge \mathtt{R} = r \wedge i < j) \;= \\
&\mathtt{sp}(\mathtt{K} := 0; \\
&\quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP}; \\
&\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J}) \\
&\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = k \wedge \mathtt{R} = 0 \wedge i < j) \;= \\
&\mathtt{sp}(\mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP}; \\
&\quad \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J}) \\
&\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge i < j) \;= \\
&(\mathtt{sp}(\mathtt{K} := \mathtt{K} + 1;\ \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J}) \\
&\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge (i < j \wedge (\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}])) \\
&\ \vee \\
&\ \mathtt{sp}(\mathtt{SKIP};\ \mathtt{IF\ K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN\ R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE\ R} := \mathtt{I} - \mathtt{J}) \\
&\quad (\exists i\ j\ k\ r.\ \mathtt{I} = i \wedge \mathtt{J} = i \wedge \mathtt{K} = 0 \wedge \mathtt{R} = 0 \wedge (i < j \wedge \neg(\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}])))
\end{aligned}
$$

Since $(\mathtt{I} < \mathtt{J})[i, j/\mathtt{I}, \mathtt{J}] = i < j$ the precondition of the second disjunct above contains the conjunct $i < j \wedge \neg(i < j)$, which is false. Thus the second disjunct can be pruned:

$$
\begin{array}{l}
(\mathtt{sp}(\mathtt{K} := \mathtt{K}+1;\ \mathtt{IF}\ \mathtt{K}=1 \wedge \neg(\mathtt{I}=\mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I}-\mathtt{J}) \\
\quad (\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=0 \wedge \mathtt{R}=0 \wedge (i<j \wedge (\mathtt{I}<\mathtt{J})[i,j/\mathtt{I},\mathtt{J}])) \\
\vee \\
\mathtt{sp}(\mathtt{SKIP};\ \mathtt{IF}\ \mathtt{K}=1 \wedge \neg(\mathtt{I}=\mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I}-\mathtt{J}) \\
\quad (\exists i\ j\ k\ r. \\
\qquad \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=0 \wedge \mathtt{R}=0 \wedge (i<j \wedge \neg(\mathtt{I}<\mathtt{J})[i,j/\mathtt{I},\mathtt{J}]))) \ = \\
\mathtt{sp}(\mathtt{K} := \mathtt{K}+1;\ \mathtt{IF}\ \mathtt{K}=1 \wedge \neg(\mathtt{I}=\mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I}-\mathtt{J}) \\
\quad (\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=0 \wedge \mathtt{R}=0 \wedge i<j])) \ = \\
\mathtt{sp}(\mathtt{IF}\ \mathtt{K}=1 \wedge \neg(\mathtt{I}=\mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J}-\mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I}-\mathtt{J}) \\
\quad (\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=(\mathtt{K}+1)[0/\mathtt{K}] \wedge \mathtt{R}=0 \wedge i<j])) \ = \\
(\mathtt{sp}(\mathtt{R} := \mathtt{J}-\mathtt{I}) \\
\quad (\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=1 \wedge \mathtt{R}=0 \wedge (i<j \wedge (1=1 \wedge \neg(i=j)))) \\
\vee \\
\mathtt{sp}(\mathtt{R} := \mathtt{I}-\mathtt{J}) \\
\quad (\exists i\ j\ k\ r. \\
\qquad \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=1 \wedge \mathtt{R}=0 \wedge (i<j \wedge \neg(1=1 \wedge \neg(i=j))))) \ =
\end{array}
$$

The second disjunct is pruned as $i<j \wedge \neg(1=1 \wedge \neg(i=j))$ simplifies to F.

$$
\begin{array}{l}
\mathtt{sp}(\mathtt{R} := \mathtt{J}-\mathtt{I}) \\
\quad (\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=1 \wedge \mathtt{R}=0 \wedge i<j) \ = \\
(\exists i\ j\ k\ r.\ \mathtt{I}=i \wedge \mathtt{J}=i \wedge \mathtt{K}=1 \wedge \mathtt{R}=(\mathtt{J}-\mathtt{I})[i,j/\mathtt{I},\mathtt{J}] \wedge i<j)
\end{array}
$$

The right hand side of this equation simplifies to $\mathtt{R}=\mathtt{J}-\mathtt{I} \wedge \mathtt{I}<\mathtt{J}$ by performing the substitution and then using properties of existential quantifiers. Thus: $\mathtt{sp\ AbsMinus}\ (\mathtt{I}<\mathtt{J}) \ = \ \mathtt{R}=\mathtt{J}-\mathtt{I} \wedge \mathtt{I}<\mathtt{J}$ by the derivation above.

A similar calculation by symbolic execution (but pruning different branches of the conditionals) gives: $\mathtt{sp\ AbsMinus}\ (\mathtt{J} \leq \mathtt{I}) \ = \ (\mathtt{R}=\mathtt{I}-\mathtt{J} \wedge \mathtt{J} \leq \mathtt{I})$.

Since $\{P\}S\{Q\}$ if and only if $(\mathtt{sp}\,S\,P) \Rightarrow Q$ it follows from the results of calculations for `AbsMinus` above that $\{\mathtt{I}<\mathtt{J}\}\mathtt{AbsMinus}\{\mathtt{R}=\mathtt{J}-\mathtt{I} \wedge \mathtt{I}<\mathtt{J}\}$ and $\{\mathtt{J} \leq \mathtt{I}\}\mathtt{AbsMinus}\{\mathtt{R}=\mathtt{I}-\mathtt{J} \wedge \mathtt{J} \leq \mathtt{I}\}$. Hence by the disjunction rule for Hoare Logic

$$\frac{\vdash \{P_1\}S\{Q_1\} \qquad \vdash \{P_2\}S\{Q_2\}}{\vdash \{P_1 \vee P_2\}S\{Q_1 \vee Q_2\}}$$

we can conclude $\vdash \{\mathsf{T}\}\mathtt{AbsMinus}\{(\mathtt{R}=\mathtt{J}-\mathtt{I} \wedge \mathtt{I}<\mathtt{J}) \vee (\mathtt{R}=\mathtt{I}-\mathtt{J} \wedge \mathtt{J} \leq \mathtt{I})\}$.

This example suggests a strategy for proving Hoare triples $\{P\}S\{Q\}$. First split P into a disjunction $P \Leftrightarrow P_1 \vee \cdots \vee P_n$ where each P_i determines a path through the program S. Then, for each i, compute $\mathtt{sp}\,P_i\,S$ by symbolic execution. Finally check that $\mathtt{sp}\,P_i\,S \Rightarrow Q$ holds for each i. If these implications all hold, then the original Hoare triple follows by the disjunction rule above. This strategy is hardly new, but explaining it as strongest postcondition calculation implemented by symbolic evaluation with Hoare logic for combining the results of the evaluations provides a nice formal foundation and also provides a link from the deductive system of Hoare logic to automated software model checking, which is often based on symbolic execution.

5 Backwards with preconditions

Calculating weakest preconditions is simpler than calculating strongest post-conditions because the assignment rules needs just one substitution and generates no additional quantifiers: $\mathtt{wp}\,(V := E)\,Q = Q[E/V]$. There is thus no need to use formulae of the form $\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi$ as one can calculate with post-conditions of any form. Furthermore, if the McCarthy conditional notation $(B \to P \mid Q)$ is defined by:

$$(B \to P \mid Q) \ = \ (P \wedge B) \vee (Q \wedge \neg B)$$

then the wp rule for conditionals can be expressed as:

$$\mathtt{wp}\,(\mathtt{IF}\,B\,\mathtt{THEN}\,S_1\,\mathtt{ELSE}\,S_2)\,Q \ = \ (B \to \mathtt{wp}\,S_1\,Q \mid \mathtt{wp}\,S_2\,Q)$$

which simplifies the calculation of wp. This is illustrated using the AbsMinus example (see Fig. 2). In the calculation that follows, assignment substitutions are performed immediately.

$$\mathtt{wp}\,\mathtt{AbsMinus}\,(\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I} < \mathtt{J}) \ =$$

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad \mathtt{IF}\ \mathtt{I} < \mathtt{J}\ \mathtt{THEN}\ \mathtt{K} := \mathtt{K} + 1\ \mathtt{ELSE}\ \mathtt{SKIP};\\ \quad \mathtt{IF}\ \mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I} - \mathtt{J})\\ \ (\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I} < \mathtt{J}) \ =\end{array}$$

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad \mathtt{IF}\ \mathtt{I} < \mathtt{J}\ \mathtt{THEN}\ \mathtt{K} := \mathtt{K} + 1\ \mathtt{ELSE}\ \mathtt{SKIP})\\ \ (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \to \ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J} \ \mid \ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J}) \ =\end{array}$$

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad (\mathtt{I}{<}\mathtt{J} \ \to \ (\mathtt{K}{+}1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \to \ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J} \ \mid \ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J})\\ \qquad\quad \mid \ (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \to \ \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J} \ \mid \ \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J})) \ =\end{array}$$

$$\begin{array}{l}(\mathtt{I}{<}\mathtt{J} \ \to \ (0{+}1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \to \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J})\\ \qquad \mid \ (0 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \to \mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J} \mid \mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J}))\end{array}$$

This calculation can be simplified on-the-fly: $(\mathtt{K}+1 = 1)$ simplifies to $(\mathtt{K} = 0)$, $(\mathtt{J}{-}\mathtt{I} = \mathtt{J}{-}\mathtt{I})$ simplifies to T and $(\mathtt{J}{-}\mathtt{I} = \mathtt{I}{-}\mathtt{J} \wedge \mathtt{I}{<}\mathtt{J})$ simplifies to F.

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad \mathtt{IF}\ \mathtt{I} < \mathtt{J}\ \mathtt{THEN}\ \mathtt{K} := \mathtt{K} + 1\ \mathtt{ELSE}\ \mathtt{SKIP};\\ \quad \mathtt{IF}\ \mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \mathtt{THEN}\ \mathtt{R} := \mathtt{J} - \mathtt{I}\ \mathtt{ELSE}\ \mathtt{R} := \mathtt{I} - \mathtt{J})\\ \ (\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I}{<}\mathtt{J}) \ =\end{array}$$

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad \mathtt{IF\ I} < \mathtt{J\ THEN\ K} := \mathtt{K} + 1\ \mathtt{ELSE\ SKIP})\\ \quad (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{I{<}J} \mid \mathsf{F}) \ =\end{array}$$

$$\begin{array}{l}\mathtt{wp}(\mathtt{R} := 0;\\ \quad \mathtt{K} := 0;\\ \quad (\mathtt{I{<}J} \ \rightarrow\ (\mathtt{K} = 0 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{I{<}J} \mid \mathsf{F})\\ \qquad\quad \mid\ (\mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \ \rightarrow\ \mathtt{I{<}J} \mid \mathsf{F})) \ =\end{array}$$

$$(\mathtt{I{<}J} \ \rightarrow\ (1 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{I{<}J} \mid \mathsf{F}) \mid (0 = 1 \wedge \neg(\mathtt{I} = \mathtt{J}) \rightarrow \mathtt{I{<}J} \mid \mathsf{F}))$$

The last formula above is $\mathtt{wp\,AbsMinus}(\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I} < \mathtt{J})$ and simplifies to $(\mathtt{I{<}J} \ \rightarrow\ \mathsf{T} \mid \mathsf{F})$, which simplifies to $\mathtt{I{<}J}$.

The Hoare triple $\{\mathtt{I{<}J}\}\mathtt{AbsMinus}\{\mathtt{R} = \mathtt{J} - \mathtt{I} \wedge \mathtt{I{<}J}\}$ then follows from $\mathtt{wp\,AbsMinus}(\mathtt{R} = \mathtt{J}{-}\mathtt{I} \wedge \mathtt{I} < \mathtt{J}) = \mathtt{I{<}J}$.

6 Forwards versus backwards

It seems clear from the example in the preceding section that proving $\{P\}S\{Q\}$ by calculating $\mathtt{wp}\,S\,Q$ is simpler than proving it by calculating $\mathtt{sp}\,S\,P$, indeed many automated Hoare logic verifiers work by calculating weakest preconditions. There are, however, several applications where strongest postconditions have a role. One such application is 'reverse engineering' where, given a precondition, one tries to deduce what a program (e.g. legacy code) does by discovering a postcondition by symbolic execution [9]. Related to this is the use of symbolic execution for testing [18]. Yet another application is to verify a given Hoare triple by symbolically executing separate paths and combining the results (this approach has already been outlined at the end of Section 2). This is a form of software model checking, where loops are unwound some number of times to create loop-free straight line code, the strongest postcondition is calculated and then an automatic tool (like an SMT solver) used to show that this entails the given postcondition.[7] The hope is that one runs enough of the program to expose significant bugs. In general, code with loops cannot be unwound to equivalent straight line code, so although this approach is a powerful bug-finding method it cannot (without further analysis) be used for full proof of correctness. An advantage of symbolic evaluation is that one can use the symbolic representation of the state-so-far to resolve conditional

[7] State of the art bounded model checkers [3, 1] generate the strongest postcondition using similar rules to those given in Fig. 1. However, they first transform programs into SSA (Static Single Assignment) form [5] and avoid the explicit use of existential quantifiers generated by assignments. The approach in this paper seems equivalent to the use of SSA, but we have not worked out a clean account of this. A feature of our method is that it applies directly to programs without requiring any preprocessing.

branches and hence prune paths. The extreme case of this is when the initial precondition specifies a unique starting state, so that symbolic execution collapses to normal 'ground' execution. Thus calculating strongest postconditions by symbolic execution provides a smooth transition from testing to full verification: by weakening the initial precondition one can make the verification cover more initial states.

7 Loops and invariants

There is no general way to calculate strongest postconditions or weakest preconditions for `WHILE`-loops. Rewriting with the equations in Fig. 1 may not terminate, so if S contains loops then the strategies for proving $\{P\}\,S\,\{Q\}$ by calculating $\mathtt{sp}\,P\,S$ or $\mathtt{wp}\,S\,Q$ will not work. Instead, we define 'approximate' versions of `sp` and `wp` called, respectively, `asp` and `awp`, together with formulae ("verification conditions") $\mathtt{svc}\,P\,S$ and $\mathtt{wvc}\,S\,Q$ with the properties that:

$$\vdash \mathtt{svc}\,S\,P \Rightarrow \{P\}\,S\,\{\mathtt{asp}\,S\,P\}$$

$$\vdash \mathtt{wvc}\,S\,Q \Rightarrow \{\mathtt{awp}\,S\,Q\}\,S\,\{Q\}$$

See Fig. 3 for equations specifying `asp` and `svc` and Fig. 4 for `awp` and `wvc`. Let ϕ be a formula and $x_1, \ldots, x_n$ all the free variables in ϕ. It is clear that if S is loop-free then $\mathtt{svc}\,S\,P$ and $\mathtt{wvc}\,S\,Q$ are true (T) and also $\mathtt{asp}\,S\,P = \mathtt{sp}\,S\,P$ and $\mathtt{awp}\,S\,Q = \mathtt{wp}\,S\,Q$. Thus for loop-free programs the 'approximate' predicate transformers are equivalent to the exact ones.

In Fig. 3, the operator `UNSAT` is true if its argument is unsatisfiable: $\mathtt{UNSAT}(\phi) = \neg\exists x_1 \cdots x_n.\ \phi$. The operator `TAUT` is true if its argument is a tautology: $\mathtt{TAUT}(\phi) = \forall x_1 \cdots x_n.\ \phi$. The relation between `UNSAT` and `TAUT` is: $\mathtt{UNSAT}(\phi) = \mathtt{TAUT}(\neg\phi)$. Point 4 of the first proof in the appendix uses the fact that $\{P\}\,S\,\{Q\}$ holds vacuously if $\mathtt{UNSAT}(P)$. Point 4 of the second proof in the appendix uses the dual fact that $\{P\}\,S\,\{Q\}$ holds vacuously if $\mathtt{TAUT}(Q)$.

To establish $\{P\}\,S\,\{Q\}$, the Hoare logic "Rules of Consequence" ensure that it is sufficient to prove either the conjunction $(\mathtt{svc}\,S\,P) \wedge (\mathtt{asp}\,S\,P \Rightarrow Q)$ or the conjunction $(\mathtt{wvc}\,S\,Q) \wedge (P \Rightarrow \mathtt{awp}\,S\,Q)$. The early development of mechanised program verification uses ideas similar to weakest preconditions to generate verification conditions [19, 17, 15, 14, 23, 20]. Strongest postconditions are less used for reasoning about programs containing loops, but generalisations of symbolic execution for them have been developed [22].

Reasoning about loops usually requires invariants to be supplied (either by a human or by some invariant-finding tool). Hoare logic provides the following `WHILE`-rule (which is a combination of Hoare's original "Rule of Iteration" and his "Rules of Consequence"). Following standard practise, we have added the invariant R as an annotation in curly brackets just before the body S of the `WHILE`-loop.

$$\frac{\vdash P \Rightarrow R \qquad \vdash \{R \wedge B\}\,S\,\{R\} \qquad R \wedge \neg B \Rightarrow Q}{\vdash \{P\}\,\mathtt{WHILE}\,B\,\mathtt{DO}\{R\}\,S\,\{Q\}}$$

$\texttt{asp}\,\texttt{SKIP}\,P = P$
$\texttt{svc}\,\texttt{SKIP}\,P = \mathsf{T}$

$\texttt{asp}\,(V := E)\,P = \exists v.\ (V{=}E[v/V]) \wedge P[v/V]$
$\texttt{svc}\,(V := E)\,P = \mathsf{T}$

$\texttt{asp}\,(S_1\,;S_2)\,P = \texttt{asp}\,S_2\,(\texttt{asp}\,S_1\,P)$
$\texttt{svc}\,(S_1\,;S_2)\,P = \texttt{svc}\,S_1\,P\ \wedge\ \texttt{svc}\,S_2\,(\texttt{asp}\,S_1\,P)$

$\texttt{asp}\,(\texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{ELSE}\,S_2)\,P = \texttt{asp}\,S_1\,(P \wedge B)\ \vee\ \texttt{asp}\,S_2\,(P \wedge \neg B)$
$\texttt{svc}\,(\texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{ELSE}\,S_2)\,P =$
$\quad(\texttt{UNSAT}(P \wedge B) \vee \texttt{svc}\,S_1\,(P \wedge B))\ \wedge\ (\texttt{UNSAT}(P \wedge \neg B) \vee \texttt{svc}\,S_2\,(P \wedge \neg B))$

$\texttt{asp}\,(\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S)\,P = R \wedge \neg B$
$\texttt{svc}\,(\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S)\,P = (P \Rightarrow R)\ \wedge\ (\texttt{asp}\,S\,(R \wedge B) \Rightarrow R)\ \wedge\ \texttt{svc}\,S\,(R \wedge B)$

Fig. 3. Approximate strongest postconditions and verification conditions

$\texttt{awp}\,\texttt{SKIP}\,Q = Q$
$\texttt{wvc}\,\texttt{SKIP}\,Q = \mathsf{T}$

$\texttt{awp}\,(V := E)\,Q = Q[E/V]$
$\texttt{wvc}\,(V := E)\,Q = \mathsf{T}$

$\texttt{awp}\,(S_1\,;S_2)\,Q = \texttt{awp}\,S_1\,(\texttt{awp}\,S_2\,Q)$
$\texttt{wvc}\,(S_1\,;S_2)\,Q = \texttt{wvc}\,S_1\,(\texttt{awp}\,S_2\,Q)\ \wedge\ \texttt{wvc}\,S_2\,Q$

$\texttt{awp}\,(\texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{ELSE}\,S_2)\,Q = (B\ \rightarrow\ \texttt{awp}\,S_1\,Q\ \mid\ \texttt{awp}\,S_2\,Q)$
$\texttt{wvc}\,(\texttt{IF}\,B\,\texttt{THEN}\,S_1\,\texttt{ELSE}\,S_2)\,Q = \texttt{TAUT}(Q)\ \vee\ (\texttt{wvc}\,S_1\,Q \wedge \texttt{wvc}\,S_2\,Q)$

$\texttt{awp}\,(\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S)\,Q = R$
$\texttt{wvc}\,(\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S)\,Q = (R \wedge B \Rightarrow \texttt{awp}\,S\,R)\ \wedge\ (R \wedge \neg B \Rightarrow Q)\ \wedge\ \texttt{wvc}\,S\,R$

Fig. 4. Approximate weakest postconditions and verification conditions

This rule is the logical basis underlying methods based on invariants for verifying WHILE-loops.

The ideas underlying asp, svc, awp and wvc are old and mostly very well known. The contribution here is a repackaging of these standard methods in a somewhat more uniform framework. The properties stated above connecting asp and svc, and awp and wvc are easily verified by structural induction. For completeness the proofs are given in an appendix.

The equations for asp in Fig 3 are the same as those for sp in Fig. 1, except for the additional equation for $\texttt{asp}\,\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S\,P$. For symbolic execution, as described in Section 4, this equation can be written as:

$$\begin{aligned}&\texttt{asp}\,(\texttt{WHILE}\,B\,\texttt{DO}\{R\}\,S)\,(\exists\overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi)\\ &\quad=\ \exists\overline{x}.\ \overline{X} = \overline{x} \wedge (R \wedge \neg B)[\overline{x}/\overline{X}]\end{aligned}$$

Thus symbolically executing $\texttt{WHILE}\, B\, \texttt{DO}\{R\}\, S$ consists in throwing away the precondition and restarting in a new symbolic state corresponding to the state specified as holding after the WHILE-loop by the Hoare rule. This is justified if the verification conditions for the WHILE-loop hold, namely:

$$\begin{aligned}
&\texttt{svc}\,(\texttt{WHILE}\, B\, \texttt{DO}\{R\}\, S)\,(\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi)\\
&\quad = \ ((\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi) \Rightarrow R)\\
&\qquad \wedge\ (\texttt{asp}\, S\,(R \wedge B) \Rightarrow R) \wedge \texttt{svc}\, S\,(R \wedge B)\\
&\quad = \ (\forall \overline{x}.\ \phi \Rightarrow R[\overline{e}/\overline{X}]) \wedge (\texttt{asp}\, S\,(R \wedge B) \Rightarrow R) \wedge \texttt{svc}\, S\,(R \wedge B)
\end{aligned}$$

This says that the precondition must entail the invariant evaluated in the state when the loop is started ($R[\overline{e}/\overline{X}]$), the invariant R must really be an invariant ($\texttt{asp}\, S\,(R \wedge B) \Rightarrow R$) and any verifications conditions when checking R is an invariant must hold ($\texttt{svc}\, S\,(R \wedge B)$). To verify that R is an invariant a recursive symbolic execution of the loop body, starting in a state satisfying $R \wedge B$, is performed. Note that:

$$\texttt{asp}\, S\,(R \wedge B) \ = \ \texttt{asp}\, S\,(\exists \overline{x}.\ \overline{X} = \overline{x} \wedge (R \wedge B)[\overline{x}/\overline{X}])$$

The equations for symbolic execution and verification conditions on symbolic state formulae are given in Fig 5. Note that $\texttt{UNSAT}(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = \texttt{UNSAT}(\phi)$.

$\texttt{asp}\, \texttt{SKIP}\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = (\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$

$\texttt{svc}\, \texttt{SKIP}\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = \mathsf{T}$

$\texttt{asp}\,(X_i := E)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$
$= \exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_i = E[\overline{e}/\overline{X}] \wedge \ldots \wedge X_n = e_n \wedge \phi$

$\texttt{svc}\,(X_i := E)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = \mathsf{T}$

$\texttt{asp}\,(S_1\,;S_2)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = \texttt{asp}\, S_2\,(\texttt{asp}\, S_1\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi))$

$\texttt{svc}\,(S_1\,;S_2)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) = \texttt{svc}\, S_1\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi) \ \wedge\ \texttt{svc}\, S_2\,(\texttt{asp}\, S_1\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi))$

$\texttt{asp}\,(\texttt{IF}\, B\, \texttt{THEN}\, S_1\, \texttt{ELSE}\, S_2)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$
$= \texttt{asp}\, S_1\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge B[\overline{e}/\overline{X}])) \ \vee\ \texttt{asp}\, S_2\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge \neg B[\overline{e}/\overline{X}]))$

$\texttt{svc}\,(\texttt{IF}\, B\, \texttt{THEN}\, S_1\, \texttt{ELSE}\, S_2)\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge \phi)$
$= (\texttt{UNSAT}(\phi \wedge B[\overline{e}/\overline{X}]) \vee \texttt{svc}\, S_1\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge B[\overline{e}/\overline{X}])))$
$\ \ \wedge$
$\ \ (\texttt{UNSAT}(\phi \wedge \neg B[\overline{e}/\overline{X}]) \vee \texttt{svc}\, S_2\,(\exists \overline{x}.\ \overline{X} = \overline{e} \wedge (\phi \wedge \neg B[\overline{e}/\overline{X}])))$

$\texttt{asp}\,(\texttt{WHILE}\, B\, \texttt{DO}\{R\}\, S)\,(\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi)$
$= \ \exists \overline{x}.\ \overline{X} = \overline{x} \wedge (R \wedge \neg B)[\overline{x}/\overline{X}]$

$\texttt{svc}\,(\texttt{WHILE}\, B\, \texttt{DO}\{R\}\, S)\,(\exists \overline{x}.\ X_1 = e_1 \wedge \ldots \wedge X_n = e_n \wedge \phi) =$
$= \ (\forall \overline{x}.\ \phi \Rightarrow R[\overline{e}/\overline{X}]) \wedge (\texttt{asp}\, S\,(R \wedge B) \Rightarrow R) \wedge \texttt{svc}\, S\,(R \wedge B)$

Fig. 5. Approximate strongest postconditions and verification conditions

As an example, consider the program Div in Fig. 6. For simplicity, assume the values of the variables in Div (i.e. R, X and ERR) are non-negative integers. First we compute asp Div (Y=0).

R:=X; Q:=0; ERR:=0;
IF Y=0 THEN ERR:=1 ELSE WHILE Y < R DO{X = R + Y × Q} (R:=R−Y;Q:=1+Q)

Fig. 6. The example program Div

Let S be WHILE Y < R DO{X = R + Y × Q} (R:=R−Y;Q:=1+Q), then:

$$
\begin{aligned}
&\texttt{asp}\\
&(\texttt{R:=X; Q:=0; ERR:=0; IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\texttt{Y=0}) =\\
&\texttt{asp}\\
&(\texttt{R:=X; Q:=0; ERR:=0; IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}q \wedge \texttt{R=}r \wedge \texttt{ERR=}e \wedge y{=}0) =\\
&\texttt{asp}\\
&(\texttt{IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}0 \wedge y{=}0) =\\
&\texttt{asp}\,(\texttt{ERR:=}1)\,(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}0 \wedge y{=}0 \wedge (y{=}0))\\
&\vee\\
&\texttt{asp}\,S\,(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}0 \wedge y{=}0 \wedge \neg(y{=}0)) \ =\\
&\texttt{asp}\,(\texttt{ERR:=}1)\,(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}0 \wedge y{=}0 \wedge (y{=}0)) \ =\\
&(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}1 \wedge y{=}0 \wedge (y{=}0)) \ =\\
&\texttt{X=R} \wedge \texttt{Y=}0 \wedge \texttt{ERR=}1
\end{aligned}
$$

Next we compute svc Div (Y=0) (simplifying $\mathsf{T} \wedge \phi$ to ϕ on-the-fly).

$$
\begin{aligned}
&\texttt{svc}\\
&(\texttt{R:=X; Q:=0; ERR:=0; IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\texttt{Y=0}) =\\
&\texttt{svc}\\
&(\texttt{R:=X; Q:=0; ERR:=0; IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}q \wedge \texttt{R=}r \wedge \texttt{ERR=}e \wedge y{=}0) =\\
&\texttt{svc}\\
&(\texttt{IF Y=0 THEN ERR:=1 ELSE}\,S)\\
&(\exists xyqre.\ \texttt{X=}x \wedge \texttt{Y=}y \wedge \texttt{Q=}0 \wedge \texttt{R=}x \wedge \texttt{ERR=}0 \wedge y{=}0) =\\
&(\texttt{UNSAT}(y = 0 \wedge y = 0) \vee \texttt{svc}\,(\texttt{ERR:=}1)\,(\cdots))\\
&\wedge\\
&(\texttt{UNSAT}(y = 0 \wedge \neg(y = 0)) \vee \texttt{svc}\,S\,(\cdots)) \ = \ (\mathsf{F} \vee \mathsf{T}) \wedge (\mathsf{T} \vee \cdots) \ = \ \mathsf{T}
\end{aligned}
$$

Thus $\{\texttt{Y=}0\}\texttt{Div}\{\texttt{X=R} \wedge \texttt{Y=}0 \wedge \texttt{ERR=}1\}$, since $\forall S\ P.\ \texttt{svc}\,S\,P \Rightarrow \{P\}\,S\,\{\texttt{asp}\,S\,P\}$. Whilst this might seem to be a very heavyweight derivation of a trivial case,

the point is that the derivation is a forward symbolic execution with some on-the-fly simplification. This simplification enables the calculation of `asp` and `svc` for the `WHILE`-loop to be avoided.
For the $\mathtt{Y} > 0$ case, it is necessary to analyse the loop, which we now do.

$$\begin{array}{l}
\mathtt{asp} \\
\ (\mathtt{R:=X;\ Q:=}0\mathtt{;\ ERR:=}0\mathtt{;\ IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\mathtt{Y}{>}0) = \\
\mathtt{asp} \\
\ (\mathtt{R:=X;\ Q:=}0\mathtt{;\ ERR:=}0\mathtt{;\ IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}q \wedge \mathtt{R}{=}r \wedge \mathtt{ERR}{=}e \wedge y{>}0) = \\
\mathtt{asp} \\
\ (\mathtt{IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0) = \\
\mathtt{asp}\,(\mathtt{ERR:=}1)\,(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0 \wedge (y{=}0)) \\
\vee \\
\mathtt{asp}\,S\,(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0 \wedge \neg(y{=}0)) \ = \\
\mathtt{asp}\,S\,(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0 \wedge \neg(y{=}0))
\end{array}$$

Since S is $\mathtt{WHILE\ Y} < \mathtt{R\ DO}\{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}\}\ (\cdots)$, it follows (see Fig. 5) that:

$$\begin{array}{l}
\mathtt{asp\ Div\ (Y}{>}0) \ = \\
\mathtt{asp}\,S\,(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0 \wedge \neg(y{=}0)) \ = \\
(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}q \wedge \mathtt{R}{=}r \wedge \mathtt{ERR}{=}e \wedge (x = r{+}y{\times}q \wedge \neg(y{<}r))) = \\
(\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} < \mathtt{R}))
\end{array}$$

Thus if $\mathtt{svc\ Div\ (Y}{>}0)$ holds (which it does, see below) then

$$\{\mathtt{Y}{>}0\}\mathtt{Div}\{\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q} \wedge \neg(\mathtt{Y} < \mathtt{R})\}$$

To verify $\mathtt{svc\ Div\ (Y}{>}0)$, first calculate:

$$\begin{array}{l}
\mathtt{svc} \\
\ (\mathtt{R:=X;\ Q:=}0\mathtt{;\ ERR:=}0\mathtt{;\ IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\mathtt{Y}{>}0) = \\
\mathtt{svc} \\
\ (\mathtt{R:=X;\ Q:=}0\mathtt{;\ ERR:=}0\mathtt{;\ IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}q \wedge \mathtt{R}{=}r \wedge \mathtt{ERR}{=}e \wedge y{>}0) = \\
\mathtt{svc} \\
\ (\mathtt{IF\ Y=}0\ \mathtt{THEN\ ERR:=}1\ \mathtt{ELSE}\,S) \\
\ (\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge y{>}0) = \\
(\mathtt{UNSAT}(y > 0 \wedge y = 0) \vee \mathtt{svc}\,(\mathtt{ERR:=}1)\,(\cdots)) \\
\wedge \\
(\mathtt{UNSAT}(y > 0 \wedge \neg(y = 0)) \vee \mathtt{svc}\,S\,(\cdots)) = (\mathsf{T} \vee \mathsf{T}) \wedge (\mathsf{F} \vee \mathtt{svc}\,\mathsf{S}\,(\cdots)) = \\
\mathtt{svc}\,S\,(\exists xyqre.\ \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{Q}{=}0 \wedge \mathtt{R}{=}x \wedge \mathtt{ERR}{=}0 \wedge (y{>}0 \wedge \neg(y = 0)))
\end{array}$$

S is a `WHILE`-loop; Fig. 5 shows the verification conditions generated:

1. $\forall x\,y\,q\,r\,e.\ (y{>}0 \wedge \neg(y = 0)) \Rightarrow (\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q})[x, y, 0, x, 0/\mathtt{X}, \mathtt{Y}, \mathtt{Q}, \mathtt{R}, \mathtt{ERR}]$
2. $\mathtt{asp}\,(\mathtt{R}{:=}\mathtt{R}{-}\mathtt{Y};\mathtt{Q}{:=}1{+}\mathtt{Q})\,((\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}) \wedge \mathtt{Y} < \mathtt{R}) \Rightarrow (\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q})$
3. $\mathtt{svc}\,(\mathtt{R}{:=}\mathtt{R}{-}\mathtt{Y};\mathtt{Q}{:=}1{+}\mathtt{Q})\,((\mathtt{X} = \mathtt{R} + \mathtt{Y} \times \mathtt{Q}) \wedge \mathtt{Y} < \mathtt{R})$

The first (1) is $(y{>}0 \wedge \neg(y = 0)) \Rightarrow (x = x + y \times 0)$, which is clearly true. The third (3) is also clearly true as $\mathtt{svc}\,S\,P = \mathsf{T}$ if S is loop-free. The second (2) requires a symbolic execution:

$$\mathtt{asp}\,(\mathtt{R}{:=}\mathtt{R}{-}\mathtt{Y};\mathtt{Q}{:=}1{+}\mathtt{Q})\,((\mathtt{X} = \mathtt{R}{+}\mathtt{Y} \times \mathtt{Q}) \wedge \mathtt{Y} < \mathtt{R}) \;=$$

$$\mathtt{asp}\ (\mathtt{R}{:=}\mathtt{R}{-}\mathtt{Y};\mathtt{Q}{:=}1{+}\mathtt{Q})\ (\exists x\ y\ q\ r.\ \mathtt{X} = x \wedge \mathtt{Y} = y \wedge \mathtt{Q} = q \wedge \mathtt{R} = r \wedge ((x = r + y{\times}q) \wedge y < r)) \;=$$

$$(\exists x\ y\ q\ r.\ \mathtt{X} = x \wedge \mathtt{Y} = y \wedge \mathtt{Q} = 1{+}q \wedge \mathtt{R} = r{-}y \wedge ((x = r + y{\times}q) \wedge y < r))$$

Thus to show verification condition 2 above, we must show:

$$(\exists x\ y\ q\ r.\ \mathtt{X} = x \wedge \mathtt{Y} = y \wedge \mathtt{Q} = 1{+}q \wedge \mathtt{R} = r{-}y \wedge ((x = r + y{\times}q) \wedge y < r))$$
$$\Rightarrow$$
$$(\mathtt{X} = \mathtt{R} + \mathtt{Y}{\times}\mathtt{Q})$$

i.e.: $(((x = r + y{\times}q) \wedge y < r)) \Rightarrow (x = (r{-}y) + y{\times}(1{+}q))$, which is true. Thus all three verification conditions, 1, 2 and 3 above, are true.

The application of weakest precondition methods (`wvc` and `awp`) to the `Div` example is completely standard and the details are well-known, so we don't give them here. However the general remarks made in Section 6 continue to apply when there are loops. In particular, using forwards symbolic execution, one can separately explore non-looping paths through a program using software model checking methods. Deductive theorem proving need only be invoked on paths with loops. The general framework based on `svc` and `asp` enables these separate analysis methods to be unified.

8 Discussion, summary and conclusion

Our goal has been to provide a review of some classical verification methods, especially Hoare logic, from the perspective of mechanical verification. We reviewed how both weakest preconditions and strongest postconditions could be used to convert the problem of verifying Hoare triples to problems in pure logic, suitable for theorem provers. Although 'going backwards' via weakest preconditions appears superficially simpler, we have tried to make a case for going forward using strongest postconditions. The benefits are that computing strongest postconditions can be formulated as symbolic execution, with loops handled via forward verification conditions. This provides a framework that can unify both deductive methods for full proof of correctness with automatic property checking based on symbolic execution.

Although the development in this paper has been based on classical Hoare logic, over the years there have been many advances that add new ideas.

Notable examples are VDM [16] that generalises postconditions to relations between the initial and final states (rather than just predicates on the final state) and separation logic [21] that provides tractable methods for handling pointers. Separation logic tools are often based on symbolic execution (though it remains to to be seen whether anything here provides a useful perspective on this).

The contribution of this paper is to explain how old methods (Floyd-Hoare logic) and new ones (software model checking) can be fitted together to provide a spectrum of verification possibilities. There are no new concepts here, but we hope to have provided a clarifying perspective that shows that Hoare's pioneering ideas will be going strong for another forty years!

References

1. J. Mantovani A. Armando and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, Feb. 2009.
2. Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey—Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.
3. Lerda F. Clarke E., Kroening D. A tool for checking ansi-c programs. In *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.
4. Stephen A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
5. Ferrante J. Cytron R., Rosen B. K., Wegman M. N., and Zadeck F. K. Efficently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13, number 4:451–490, 1991.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.
7. Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and Monographs in Computer Science. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
8. Robert W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967.
9. G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 188, Washington, DC, USA, 1995. IEEE Computer Society.
10. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
11. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 232–244. ACM, 2004.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
13. C. A. R. Hoare and Peter E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Inf.*, 3:135–153, 1974.

14. S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.
15. Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification i: a logical basis and its implementation. Technical report, Stanford University, Stanford, CA, USA, 1973.
16. Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Ann. Hist. Comput.*, 25(2):26–49, 2003.
17. James C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.
18. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
19. James Cornelius King. *A program verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.
20. David C. Luckham. A brief account: Implementation and applications of a pascal program verifier (position statement). In *ACM '78: Proceedings of the 1978 annual conference*, pages 786–792, New York, NY, USA, 1978. ACM.
21. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, September 2001.
22. Corina S. Pasareanu and Willem Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.
23. F. W von Henke and D. C. Luckham. A methodology for verifying programs. In *Proceedings of the international conference on Reliable software*, pages 156–164, New York, NY, USA, 1975. ACM.
24. Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

Appendix: proofs relating svc, asp, wvc, awp

Proof by structural induction on S that $\forall S\ P.\ \texttt{svc}\ S\ P \Rightarrow \{P\}\,S\,\{\texttt{asp}\ S\ P\}$

1. SKIP.
   Follows from $\vdash\ \{P\}\,\texttt{SKIP}\,\{P\}$.
2. $V := E$.
   Follows from $\vdash\ \{P\}\,V := E\,\{\exists v.\ (V = E[v/V]) \wedge P[v/V]\}$.
3. $S_1\,;S_2$.
   Assume by induction:
   $\forall P.\ \texttt{svc}\ S_1\ P \Rightarrow \{P\}\,S_1\,\{\texttt{asp}\ S_1\ P\}$
   $\forall P.\ \texttt{svc}\ S_2\ P \Rightarrow \{P\}\,S_2\,\{\texttt{asp}\ S_2\ P\}$
   Specialising P to $\texttt{asp}\ S_1\ P$ in the inductive assumption for S_2 yields:
   $\texttt{svc}\ S_2\ (\texttt{asp}\ S_1\ P) \Rightarrow \{\texttt{asp}\ S_1\ P\}\,S_2\,\{\texttt{asp}\ S_2\ (\texttt{asp}\ S_1\ P)\}$
   Hence by inductive assumption for S_1 and the Hoare sequencing rule:
   $\texttt{svc}\ S_1\ P \wedge \texttt{svc}\ S_2\ (\texttt{asp}\ S_1\ P) \Rightarrow \{P\}\,S_1\,;S_2\,\{\texttt{asp}\ S_2\ (\texttt{asp}\ S_1\ P)\}$
   Hence by definitions of $\texttt{svc}\ (S_1\,;S_2)\ P$ and $\texttt{asp}\ (S_1\,;S_2)\ P$:
   $\texttt{svc}\ (S_1\,;S_2)\ P \Rightarrow \{P\}\,S_1\,;S_2\,\{\texttt{asp}\ (S_1\,;S_2)\ P\}$.
4. IF B THEN S_1 ELSE S_2.
   Assume by induction:
   $\forall P.\ \texttt{svc}\ S_1\ P \Rightarrow \{P\}\,S_1\,\{\texttt{asp}\ S_1\ P\}$
   $\forall P.\ \texttt{svc}\ S_2\ P \Rightarrow \{P\}\,S_2\,\{\texttt{asp}\ S_2\ P\}$
   Specialising these with $P \wedge B$ and $P \wedge \neg B$, respectively, yields:
   $\texttt{svc}\ S_1\ (P \wedge B) \Rightarrow \{P \wedge B\}\,S_1\,\{\texttt{asp}\ S_1\ (P \wedge B)\}$
   $\texttt{svc}\ S_2\ (P \wedge \neg B) \Rightarrow \{P \wedge \neg B\}\,S_2\,\{\texttt{asp}\ S_2\ (P \wedge \neg B)\}$
   Applying the 'postcondition weakening' Hoare logic Rule of Consequence:
   $\texttt{svc}\ S_1\ (P \wedge B) \Rightarrow \{P \wedge B\}\,S_1\,\{\texttt{asp}\ S_1\ (P \wedge B) \vee \texttt{asp}\ S_2\ (P \wedge \neg B)\}$
   $\texttt{svc}\ S_2\ (P \wedge \neg B) \Rightarrow \{P \wedge \neg B\}\,S_2\,\{\texttt{asp}\ S_1\ (P \wedge B) \vee \texttt{asp}\ S_2\ (P \wedge \neg B)\}$
   Hence by definition of $\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P$:
   $\texttt{svc}\ S_1\ (P \wedge B) \Rightarrow \{P \wedge B\}\,S_1\,\{\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P\}$
   $\texttt{svc}\ S_2\ (P \wedge \neg B) \Rightarrow \{P \wedge \neg B\}\,S_2\,\{\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P\}$
   Since Hoare triples are true if the precondition is unsatisfiable:
   $\texttt{UNSAT}(P \wedge B) \Rightarrow \{P \wedge B\}\,S_1\,\{\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P\}$
   $\texttt{UNSAT}(P \wedge \neg B) \Rightarrow \{P \wedge \neg B\}\,S_2\,\{\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P\}$
   By definition of $\texttt{svc}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P$ and Hoare logic rules:
   $\texttt{svc}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P$
   $\Rightarrow \{P\}\,\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\,\{\texttt{asp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ P\}$
5. WHILE B DO$\{R\}$ S.
   Assume by induction:
   $\forall P.\ \texttt{svc}\ S\ P \Rightarrow \{P\}\,S\,\{\texttt{asp}\ S\ P\}$
   Specialise P to $R \wedge B$:
   $\texttt{svc}\ S\ (R \wedge B) \Rightarrow \{R \wedge B\}\,S\,\{\texttt{asp}\ S\ (R \wedge B)\}$
   By definition of $\texttt{svc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P$ and Consequence Rules:
   $\texttt{svc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P \Rightarrow \{R \wedge B\}\,S\,\{R\}$
   By definition of $\texttt{svc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P$ and Hoare WHILE-rule:
   $\texttt{svc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P \Rightarrow \{P\}\,\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S\,\{R \wedge \neg B\}$
   Hence by definition of $\texttt{asp}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P$:
   $\texttt{svc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P \Rightarrow \{P\}\,\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S\,\{\texttt{asp}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ P\}$

Proof by structural induction on S that:
$\forall S\ Q.\ \texttt{wvc}\ S\ Q \Rightarrow \{\texttt{awp}\ S\ Q\}\ S\ \{Q\}$

1. $\texttt{SKIP}$.
   Follows from $\vdash\ \{Q\}\,\texttt{SKIP}\,\{Q\}$.

2. $V := E$.
   Follows from $\vdash\ \{Q[E/V]\}\,V := E\,\{Q\}$.

3. $S_1\,;S_2$.
   Assume by induction:
   $\forall Q.\ \texttt{wvc}\ S_1\ Q \Rightarrow \{\texttt{awp}\ S_1\ Q\}\ S_1\ \{Q\}$
   $\forall Q.\ \texttt{wvc}\ S_2\ Q \Rightarrow \{\texttt{awp}\ S_2\ Q\}\ S_2\ \{Q\}$
   Specialising Q to $\texttt{awp}\ S_2\ Q$ in the inductive assumption for S_1 yields:
   $\texttt{wvc}\ S_1\ (\texttt{awp}\ S_2\ Q) \Rightarrow \{\texttt{awp}\ S_1\ (\texttt{awp}\ S_2\ Q)\}\ S_1\ \{\texttt{awp}\ S_2\ Q\}$
   Hence by inductive assumption for S_2 and the Hoare sequencing rule:
   $\texttt{wvc}\ S_1\ (\texttt{awp}\ S_2\ Q) \wedge \texttt{wvc}\ S_2\ Q \Rightarrow \{\texttt{awp}\ S_1\ (\texttt{awp}\ S_2\ Q)\}\ S_1\,;S_2\ \{Q\}$
   Hence by definitions of $\texttt{wvc}\ (S_1\,;S_2)\ Q$ and $\texttt{awp}\ (S_1\,;S_2)\ Q$:
   $\texttt{wvc}\ (S_1\,;S_2)\ Q \Rightarrow \{\texttt{awp}\ (S_1\,;S_2)\ Q\}\ S_1\,;S_2\ \{Q\}$.

4. $\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2$.
   Assume by induction:
   $\texttt{wvc}\ S_1\ Q \Rightarrow \{\texttt{awp}\ S_1\ Q\}\ S_1\ \{Q\}$
   $\texttt{wvc}\ S_2\ Q \Rightarrow \{\texttt{awp}\ S_2\ Q\}\ S_2\ \{Q\}$
   Strengthening the preconditions using the Rules of Consequence
   $\texttt{wvc}\ S_1\ Q \Rightarrow \{\texttt{awp}\ S_1\ Q \wedge B\}\ S_1\ \{Q\}$
   $\texttt{wvc}\ S_2\ Q \Rightarrow \{\texttt{awp}\ S_2\ Q \wedge \neg B\}\ S_2\ \{Q\}$
   Rewriting the preconditions using $\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q$
   $\texttt{wvc}\ S_1\ Q \Rightarrow \{\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q \wedge B\}\ S_1\ \{Q\}$
   $\texttt{wvc}\ S_2\ Q \Rightarrow \{\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q \wedge \neg B\}\ S_2\ \{Q\}$
   Since Hoare triples are true if the postcondition is a tautology
   $\texttt{TAUT}(Q) \Rightarrow \{\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q \wedge B\}\ S_1\ \{Q\}$
   $\texttt{TAUT}(Q) \Rightarrow \{\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q \wedge \neg B\}\ S_2\ \{Q\}$
   By the definition of $\texttt{wvc}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q$ and the Hoare conditional rule
   $\texttt{wvc}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q$
   $\Rightarrow \{\texttt{awp}\ (\texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2)\ Q\}\ \texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \{Q\}$

5. $\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S$.
   Assume by induction:
   $\forall Q.\ \texttt{wvc}\ S\ Q \Rightarrow \{\texttt{awp}\ S\ Q\}\ S\ \{Q\}$
   Specialise Q to R. By $\texttt{wvc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ Q$ and Consequence Rule:
   $\texttt{wvc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ Q \Rightarrow \{R \wedge B\}\ S\ \{R\}$
   By Hoare WHILE-rule:
   $\texttt{wvc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ Q \Rightarrow \{R\}\ \texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S\ \{R \wedge \neg B\}$
   By definitions of awp and wvc for $\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S$, and Consequence Rule:
   $\texttt{wvc}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ Q \Rightarrow \{\texttt{awp}\ (\texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S)\ Q\}\ \texttt{WHILE}\ B\ \texttt{DO}\{R\}\ S\ \{Q\}$

Bibliographie

[1] H. Amjad : LCF-style propositional simplification with BDDs and SAT solvers. *In Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 de *LNCS*. Springer-Verlag, 2008.

[2] Bruno Barras : Programming and computing in HOL. *In Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 de *LNCS*, pages 17–37. Springer-Verlag, 2000.

[3] G. Barthe, Burdy, L. Charles, J. Gregoire, B. Huisman, M. Lanet, J.-L. Pavlova et A. M. Requet : JACK-a tool for validation of security and behaviour of java applications. *In 5th International Symposium on Formal Methods for Components and Objects*, volume 4709 de *LNCS*, pages 152–174. Springer-Verlag, 2008.

[4] H. Collavizza et M.J.C Gordon : Semantically-driven bounded model checking using theorem proving, SMT and constraint solving. Rapport technique I3S/RR-2009-13-FR, Laboratoire I3S, Université de Nice – Sophia Antipolis, septembre 2009.

[5] A.C.J Fox : Formal specification and verification of ARM6. *In Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 de *LNCS*, pages 25–40. Springer-Verlag, 2003.

[6] M.J.C. Gordon : *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, chapitre From LCF to HOL : a short history, pages 169–186. Foundations of computing. MIT Press, 2000.

[7] M.J.C Gordon et H. Collavizza : Forward with Hoare. *In Reflections on the Work of C.A.R. Hoare*, History of Computing Series (to appear). Springer, 2010.

[8] C. A. R. Hoare : An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[9] Magnus O. Myreen et Michael J.C. Gordon : Verified lisp implementations on ARM, x86 and PowerPC. *In à paraitre TPHOL*, 2009.

[10] Matthew J. Parkinson et Gavin M. Bierman : Separation logic, abstraction and inheritance. *In POPL*, pages 75–86. ACM, 2008.

[11] L. C. Paulson : The relative consistency of the axiom of choice mechanized using isabelle/zf. *Journal of Computation and Mathematics*, 6:198–248, 2003.

[12] Henzinger Thomas A., Ranjit Jhala, Rupak Majumdar et McMillan Kenneth L. : Abstractions from proofs. *In POPL*, pages 232–244. ACM, 2004.

[13] T. Tuerk : A separation logic framework in HOL. *In Theorem Proving in Higher Order Logics (TPHOLs) - Emerging Trends*, volume 5170 de *LNCS*. Springer-Verlag, 2008.

[14] Tjark Weber et Hasan Amjad : Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic,Special Issue : Empirically Successful Computerized Reasoning*, 7(1):26–40, March 2009.

Chapitre V

Contraintes sur domaines continus

Ce chapitre présente mes travaux sur les contraintes en domaines continus, effectués en collaboration avec Michel Rueher avec lequel j'ai co-encadré le DEA puis la thèse de doctorat de François Delobel[1]. Il s'agit de deux types de travaux : un travail de synthèse qui compare les deux types de consistances partielles les plus usitées à l'époque de ces travaux (années 1997-2000) et une application pratique originale, qui consiste à étendre le domaine d'une solution initiale.

Les travaux sur la comparaison des consistances ont été présentés à la conférence CP'97 [11] et publiés dans le journal "Reliable Computing" [12] ; l'extension du domaine d'une variable a été présentée à la conférence IJCAI'99 [13]. J'ai choisi de n'introduire dans ce chapitre que les définitions et éléments nécessaires à la compréhension de mes contributions. Les preuves peuvent être trouvées dans les articles supports sections V.5.1 et V.5.2 ainsi que dans la thèse de François Delobel [15] qui a implémenté les différentes consistances partielles utilisées.

Ce chapitre est organisé comme suit. Je présente tout d'abord les contraintes sur domaines continus, puis la comparaison entre les consistances basées sur la 2B–consistance et la box–consistance. Je détaille ensuite les travaux sur la consistance intérieure et l'extension d'une solution initale. Je conclus enfin par une discussion sur les solveurs de contraintes en domaines continus.

V.1 Contraintes sur domaines continus

V.1.1 Problématique

Les systèmes de contraintes sur les nombres réels (ou contraintes sur *domaines continus* par opposition aux *domaines finis*) sont indispensables pour modéliser bon nombres d'applications. Par exemple, de telles contraintes peuvent modéliser les lois fondamentales en physique comme la loi d'ohm $U = R*I$, en balistique pour calculer l'intervalle dans lequel est compris le point d'impact au sol, ou bien en mécanique, pour modéliser un système de frein ABS[2].

Plusieurs approches ont été employées pour résoudre de tels problèmes, comme en particulier les méthodes basées sur l'analyse des intervalles [38, 37]. Nous nous intéressons ici aux algorithmes de résolution de contraintes qui, comme dans le cas des domaines finis, combinent des étapes de filtrage et de recherche (voir chapitre III section III.2.1 page 94). Ces algorithmes ont été implémentés avec succès dans les systèmes précurseurs `Newton` et `Numerica` [2, 23], et ont permis de résoudre des systèmes de contraintes sur les réels non triviaux. Actuellement, la plupart des solveurs en domaines continus combinent des techniques de filtrage et des méthodes d'analyse d'intervalle (voir par exemple les solveurs `RealPaver` ou `GlobSol` [27, 21]).

Notre étude concerne les *consistances partielles* nécessaires à l'étape de filtrage. Dans le cas des domaines finis, les consistances partielles sont dérivées de l'arc–

[1] J'ai également co-encadré le DEA de Vincent Gay-Para avec Michel Rueher et Olivier Meste, alors Maître de Conférences à Polytech et spécialiste de traitement de signal bio-médical. Il s'agissait d'utiliser les CSP en domaines continus pour analyser une série de mesures en fonction de stimuli d'entrée, l'idée étant de transformer l'ensemble discret des observations en une courbe continue plus simple à analyser. Ces travaux n'ont pas été poursuivis suite au recrutement de Vincent comme ingénieur.

[2] Ceci est un des cas d'étude du projet TESTEC sur le test des programmes temps réels.

consistance qui consiste à chercher, pour une contrainte et une variable donnée de cette contrainte, des supports dans le domaine des autres variables (voir III.2.1 page 94). Dans le cas continu, la définition de ces consistances fait appel à *l'arithmétique des intervalles*. Le point clef de l'arithmétique des intervalles est d'approximer les nombres réels par des intervalles pour quantifier les erreurs de calcul introduites par la représentation machine des nombres réels avec une précision finie [2]. Même si le problème initial est formulé en terme de contraintes sur les *nombres réels* [3], les consistances sont définies sur des intervalles qui représentent un ensemble continu de nombres réels compris entre deux nombres *flottants* (les deux bornes de l'intervalle). De façon grossière, les consistances partielles pour les domaines continus assurent que les bornes des intervalles ont des supports dans le domaine des autres variables.

La suite de cette sous-section introduit les notations pour distinguer réels, intervalles et flottants, donne un ensemble de définitions sur l'arithmétique des intervalles et enfin définit les CSP en domaines continus.

V.1.2 Notations

Les fonctions sur les nombres réels $\mathbb{R}^n \to \mathbb{R}$ seront notées f, g, h et une contrainte sur les réels $\mathbb{R}^n \to \mathbb{B}$ sera notée c. Comme dans le cas des domaines finis, $Var(c)$ représente les variables qui interviennent dans la contrainte c.

Pour distinguer les intervalles des réels, nous emploierons principalement les notations suggérées par Kearfott [26]. Les caractères gras représenteront des intervalles, les minuscules représenteront des quantités scalaires et les majuscules des vecteurs ou des ensembles. Les crochets « [.] » délimiteront des intervalles tandis que les parenthèses « (.) » délimiteront des vecteurs. Les lettres soulignées représenteront les bornes inférieures des intervalles et les lettres sur-lignées représenteront les bornes supérieures des intervalles. Ainsi, l'intervalle $\mathbf{x}$ sera représenté par $[\underline{x}, \overline{x}]$. $\tilde{x}$ représente n'importe quelle valeur dans l'intervalle $\mathbf{x}$ (en général il ne s'agit *pas* du centre de $\mathbf{x}$). De façon similaire, $\tilde{X}$ représente n'importe quel tuple de valeurs dans le vecteur d'intervalles $\mathbf{X}$. $\mathcal{I}$ représente l'ensemble des intervalles et il est ordonné par l'inclusion ensembliste.

Nous emploierons également les notations suivantes, qui ne sont pas tout à fait standard, mais sont nécessaires pour analyser finement la mise en œuvre des algorithmes :

- $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, +\infty\}$ représente l'ensemble des nombres réels augmenté des symboles d'infinité $\{-\infty, +\infty\}$.
- $\overline{\mathbb{F}}$ représente un sous-ensemble *fini* de $\mathbb{R}^\infty$ qui contient $\{-\infty, +\infty\}$. En pratique, $\overline{\mathbb{F}}$ correspond à l'ensemble des nombres flottants utilisés dans l'implémentation du solveur de contraintes.
- Si a est une constante dans $\overline{\mathbb{F}}$, a^+ (resp. a^-) correspond au plus petit (resp. grand) nombre dans $\overline{\mathbb{F}}$ strictement plus grand (resp. petit) que a.

V.1.3 Arithmétique des intervalles

Cette sous-section introduit les bases de l'arithmétique des intervalles.

[3] en supposant que l'utilisateur ne se préoccupe pas des problèmes d'implémentation en machine.

Définition V.1.1 (Intervalle)
Un intervalle $\mathbf{x} = [\underline{x}, \overline{x}]$*, avec* $\underline{x}$ *et* $\overline{x} \in \overline{I\!F}$*, est l'ensemble des nombres réels* $\{r \in \mathbb{R} \mid \underline{x} \leq r \leq \overline{x}\}$

Définition V.1.2 (Extension d'ensemble)
Soit S *un sous-ensemble de* $\mathbb{R}$*. L'*intervalle englobant *de* S *("hull" en anglais), noté* $\Box S$*, est le plus petit intervalle* I *tel que* $S \subseteq I$*.*

Remarque sur les opérations flottantes Le terme «plus petit sous-ensemble» (par rapport à l'inclusion) doit être compris ici en fonction de la précision des opérations flottantes. Dans le reste du chapitre, nous considérons comme dans [2, 30] que les résultats des opérations flottantes sont arrondis vers l'extérieur pour préserver l'exactitude du calcul. Nous supposerons également que la plus grande erreur de calcul est toujours inférieure à un flottant.
Une conséquence de ces hypothèses est que nous utiliserons respectivement les intervalles $[\underline{x}, \underline{x}^+[$, et $]\overline{x}^-, \overline{x}]$ pour désigner le plus petit intervalle qui contient le nombre réel le plus proche de $\underline{x}$ (resp. $\overline{x}$).

Définition V.1.3 (Extension aux intervalles [22, 37])
$\mathbf{f} : \mathcal{I}^n \to \mathcal{I}$ *est une* **extension aux intervalles** *de* $f : \mathbb{R}^n \to \mathbb{R}$ *ssi*
$\forall \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{I} : f(\tilde{x}_1, \ldots, \tilde{x}_n) \in \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$.
$\mathbf{c} : \mathcal{I}^n \to \mathbb{B}$ *est une* **extension aux intervalles** *de* $c : \mathbb{R}^n \to \mathbb{B}$ *ssi*
$\forall\ \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{I} : c(\tilde{x}_1, \ldots, \tilde{x}_n) \Rightarrow \mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_n})$

Définition V.1.4 (Extension optimale aux intervalles)
Soit $\mathbf{f}$ *une extension aux intervalles de* f*.* $\mathbf{f}$ *est* **optimale** *ssi*
$\forall \mathbf{X} \in \mathcal{I}^n, \mathbf{f}(\mathbf{X}) = \Box \{f(\tilde{X}) / \tilde{X} \in \mathbf{X}\}$

Les extensions optimales aux intervalles peuvent être définies pour quasiment toutes les fonctions élémentaires. L'exemple suivant définit les extensions optimales des opérations arithmétique $+, -, \times, /$.

Exemple V.1 (Extensions optimales de $+, -, \times, /$) Les extensions optimales des opérations élémentaires $+, -, \times, /$, notées $\oplus, \ominus, \otimes, \oslash$ dans la suite, sont définies par :
$[\underline{x}, \overline{x}] \oplus [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
$[\underline{x}, \overline{x}] \ominus [\underline{y}, \overline{y}] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$
$[\underline{x}, \overline{x}] \otimes [\underline{y}, \overline{y}] = [min(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y}), max(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y})]$
$[\underline{x}, \overline{x}] \oslash [\underline{y}, \overline{y}] = [min(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}), max(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y})] si\ 0 \notin [\underline{y}, \overline{y}]$ ♯

Définition V.1.5 (Extension naturelle aux intervalles [37])
$\mathbf{f}$ *est l'extension* **naturelle** *aux intervalles de* f *ssi* $\mathbf{f}$ *est obtenue en remplaçant dans* f *:*
- *chaque constante* k *avec le plus petit intervalle contenant* k*,*
- *chaque variable* x *avec la variable d'intervalle* $\mathbf{x}$*, et*
- *chaque opération avec son extension* optimale *aux intervalles.*

Dans le reste du chapitre, **c** représentera l'extension *naturelle* aux intervalles de c.

L'intérêt majeur de l'extension *naturelle* aux intervalles est qu'elle est très facilement calculée à partir des extensions optimales des opérations élémentaires. L'exemple suivant, tiré du cours sur la programmation par contraintes de Michel Rueher [42], montre que la forme initiale de l'expression associée à la fonction f à un fort impact sur la précision de son extension naturelle, puisque celle-ci est une substitution purement syntaxique.

Exemple V.2 (Extension naturelle de $x^2 - x$ et $x(x-1)$) Soit $f_1(x) = x^2 - x$ et $f_2(x) = x(x-1)$ deux formes différentes de la même expression sur les réels. Alors
$\mathbf{f}_1([0,5]) = [\ 5, 25]$
$\mathbf{f}_2([0,5]) = [\ 5, 20]$
De plus, la valeur de l'extension optimale aux intervalles de f_1 et f_2 sur $[0,5]$ est $[0.25, 20]$ puisque la dérivée de f_1 et f_2 s'annule pour x = 0.5. ♯

Nous rappelons maintenant un résultat fondamental de l'analyse des intervalles qui a de nombreuses conséquences sur l'efficacité et la précision des techniques de résolution de contraintes. En particulier, il est à la base des résultats que nous avons établis pour comparer la 2B–consistance et la box–consistance.

Proposition V.1.1 *[37] Soit $\mathbf{f} : \mathcal{I}^n \to \mathcal{I}$ l'extension naturelle aux intervalles de $f : \mathbb{R}^n \to \mathbb{R}$. Si chaque variable x_i apparaît une seule fois dans f alors $\Box\{f(\tilde{x}_1, \ldots, \tilde{x}_n)\} = \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$ sinon $\Box\{f(\tilde{x}_1, \ldots, \tilde{x}_n)\} \subseteq \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$.*

Ce résultat peut être trivialement étendu aux relations sur $\mathbb{R}^n$:

Proposition V.1.2 *Soit $\mathbf{c} : \mathcal{I}^n \to \mathbb{B}$ l'extension naturelle aux intervalles de $c : \mathbb{R}^n \to \mathbb{B}$, si chaque variable x_i apparaît une seule fois dans c, alors $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_n}) \Leftrightarrow c(\tilde{x}_1, \ldots, \tilde{x}_n)$.*

Les deux propositions ci-dessus montrent qu'excepté le cas très particulier où les variables apparaissent une seule fois dans les expressions, utiliser l'extension *naturelle* aux intervalles donne une boîte englobante trop grande et donc imprécise. Ceci est illustré sur l'exemple suivant.

Exemple V.3 (Surestimation de l'extension naturelle) Soit $f(x) = x - x$, alors $\mathbf{f}(\mathbf{x}) = [\underline{x} - \overline{x}, \overline{x} - \underline{x}]$. Par conséquent, plus l'intervalle auquel s'applique la fonction **f** est grand, plus l'intervalle résultat est grand. Par exemple, $\mathbf{f}[-30, 60] = [-90, 90]$. ♯

V.1.4 Système de contraintes sur domaines continus

Définition V.1.6 (CSP)
Un **CSP sur domaines continus** *[34] est un couple (X, C) où $X = \{x_1, \ldots, x_n\}$ représente un ensemble de variables avec pour domaines associés les intervalles $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$, et $C = \{c_1, \ldots, c_m\}$ représente un ensemble de contraintes.*

Cette définition est analogue au cas des domaines finis (voir section III.2.1 page 94) mais les notations X et $\mathbf{X}$ permettent de ne pas mentionner de façon explicite l'ensemble des variables X dans la définition du CSP.

Notations

- $P_\emptyset$ représente un CSP vide c'est-à-dire un CSP dont au moins un des domaines est vide.
- La notation $\mathbf{X}' \subseteq \mathbf{X}$ signifie que $\mathbf{x'_i} \subseteq \mathbf{x_i}$ pour tout i.
- Un CSP $P = (X, C)$ est plus petit qu'un CSP $P' = (X', C)$ ssi $\mathbf{X} \subseteq \mathbf{X}'$. Nous notons $P \preceq P'$ cette relation. Par convention, $P_\emptyset$ est le plus petit CSP.

V.1.5 Contributions

À l'époque de ces travaux (années 1997-2000), deux grandes approches existaient pour résoudre les CSP en domaines continus. Les approches basées sur les consistances de type "box–consistance" [2, 25] qui font appel à des méthodes de calcul numérique (i.e. extension de la méthode de Newton aux intervalles), et les approches basées sur la "2B–consistance" qui travaillent sur les bornes des intervalles [10, 6, 30, 32].

> Nos contributions à la résolution de contraintes en domaines continus portent sur deux points :
>
> 1. la comparaison fine des familles de consistance de la 2B–consistance et de la box–consistance,
> 2. la formalisation de la consistance intérieure et de l'extension du domaine d'une variable à partir d'un ensemble connu de solutions ; la mise en œuvre efficace de l'approche proposée emploie la box–consistance pour chercher la borne maximale de cette extension.
>
> La première contribution a permis de comparer la qualité du filtrage associé aux consistances, et ce de façon précise, en tenant compte des impératifs de mise en œuvre. La deuxième contribution a été une nouvelle application des CSP en domaines continus ; de plus, elle a fait un usage original et pertinent de la box–consistance.

V.2 Comparaison de consistances partielles

Comme expliqué section III.2.1, les solveurs de contraintes sur domaines finis sont basés sur la consistance locale d'*arc–consistance* [34, 44]. De la même manière, de nombreux solveurs sur domaines continus [40, 23, 4] sont basés sur des *relaxations* de l'arc–consistance.

La 2B–consistance [10, 6, 30, 32] est une relaxation de l'arc–consistance qui consiste à vérifier la propriété d'arc–consistance uniquement sur les deux bornes de l'intervalle. En pratique, la 2B–consistance nécessite de calculer les fonctions de projection de la contrainte sur une des variables. Cela implique de décomposer le système en contraintes primitives (i.e. monotones ou décomposables en parties monotones).

La box–consistance [2, 25] est une relaxation de l'arc–consistance plus large que la 2B–consistance. La variable x est *box–consistante* pour la contrainte c, si les bornes du domaine de x correspondent au zéro le plus à gauche et le plus à droite de

l'extension optimale aux intervalles de c. En pratique, la box–consistance utilise la méthode de Newton étendue aux intervalles pour chercher ces zéros.

La 3B–consistance et la bound–consistance sont des extensions d'ordre supérieur de la 2B–consistance et de la box–consistance[4] qui ont été introduites pour limiter les effets d'un traitement strictement local :

- La *3B–consistance* [32] est une relaxation de la consistance de chemin [17], qui est elle-même une extension d'ordre supérieur de l'arc-consistance. La 3B–consistance vérifie la 2B–consistance quand le domaine d'une des variables est réduit à la valeur d'une de ses bornes dans le système tout entier ;
- La *bound-consistance* [23, 41] applique le principe de la 3B–consistance à la box–consistance : elle vérifie la box–consistance quand le domaine d'une des variables est réduit à la valeur d'une de ses bornes dans le système tout entier.

Les sous-sections suivantes définissent ces consistances et énoncent les propriétés de comparaison.

V.2.1 Définition des consistances

2B–consistance

La 2B–consistance établit une propriété sur les bornes du domaine d'une variable de façon locale à une contrainte.

Définition V.2.1 (2B–Consistance [32])
Soit (X, C) un CSP et $c \in C$ une contrainte k-aire sur $(x_1, \ldots, x_k)$. c est 2B–consistante ssi :
$\forall i, \mathbf{x_i} = \square\{\tilde{x}_i \in \mathbf{x_i} \mid \exists \tilde{x}_1 \in \mathbf{x_1}, \ldots, \exists \tilde{x}_{i-1} \in \mathbf{x_{i-1}}, \exists \tilde{x}_{i+1} \in \mathbf{x_{i+1}}, \ldots, \exists \tilde{x}_k \in \mathbf{x_k}$ *tel que* $c(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i, \tilde{x}_{i+1} \ldots, \tilde{x}_k)\}$.
Un CSP est 2B–consistant ssi toutes ses contraintes sont 2B–consistantes.

De façon informelle, cette définition exprime que c est 2B–consistante si, pour n'importe quelle variable $x_i \in Var(c)$, il existe des valeurs dans les domaines des autres variables de $Var(c)$ qui satisfont c quand x_i est fixé à $\underline{x}$ et à $\overline{x}$. En effet, $\square S$ est le plus petit intervalle contenant S (voir définition V.1.2). Donc si $\mathbf{x_i} = \square S$ avec S de la forme $S = \{\tilde{x}_i \in \mathbf{x_i} | P(\tilde{x}_i)\}$, cela signifie que $\underline{x_i}$ et $\overline{x_i}$ vérifient la propriété P.

Nous notons $\Phi_{2B}(P)$ le filtrage par 2B–consistance de P. Dans la suite nous utiliserons le terme de *fermeture par 2B–consistance*, puisque ce filtrage existe toujours et est unique [32].

D'un point de vue géométrique, calculer un intervalle 2B–consistant pour une variable x_i revient à calculer le plus petit intervalle qui englobe la projection de la contrainte sur cette variable. Ceci est illustré sur la figure V.1.

[4] "ordre supérieur" signifie ici que la *3B–consistance* et la *bound–consistance* utilisent la *2B–consistance* et la *box–consistance* comme briques primitives.

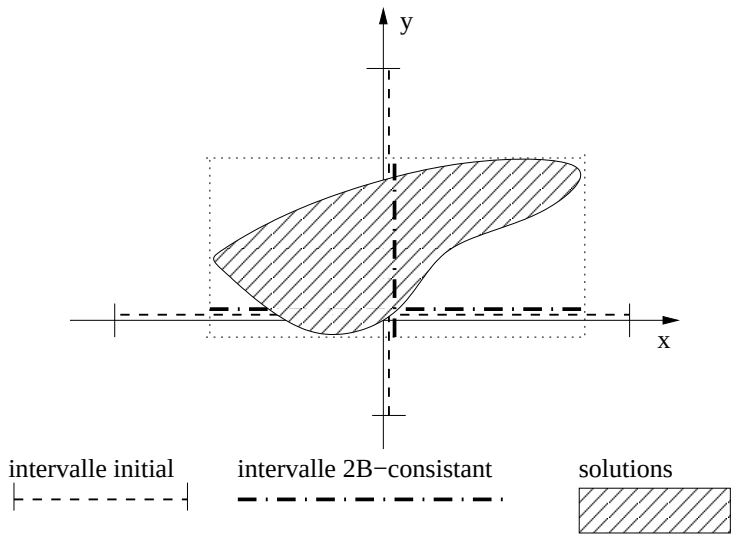


FIG. V.1 – Intervalles 2B–consistants pour les variables x et y

Calcul de la 2B–consistance

Le calcul de la fermeture par 2B–consistance est effectué par un algorithme de point fixe similaire à celui qui assure l'arc–consistance dans le cas des domaines finis (voir section III.2.1 figure III.2). Dans ce cas, la fonction de filtrage réduit les domaines des variables jusqu'à ce qu'ils soient 2B–consistants.

En pratique, l'outil de base pour l'opération de filtrage est l'approximation des fonctions de projection. Soit c une contrainte k-aire sur $X = (x_1, \ldots, x_k)$. Pour tout i dans $1..k$, $\pi_i(c, \mathbf{X})$ représente la projection sur x_i des solutions de c dans l'espace délimité par $\mathbf{X}$. Cette projection est approximée par un intervalle.

Définition V.2.2 (Approximation des fonctions de projection)
$AP_i(c, \mathbf{X}) : (C, \mathcal{I}^k) \rightarrow \mathcal{I}$ *est une* **approximation** *de* $\pi_i(c, \mathbf{X})$ *ssi*
$AP_i(c, \mathbf{X}) = \square\, \pi_i(c, \mathbf{X}) = [\min\ \pi_i(c, \mathbf{X}), \max\ \pi_i(c, \mathbf{X})]$.

En d'autres termes, $AP_i(c, \mathbf{X})$ est le plus petit intervalle qui encadre la projection $\pi_i(c, \mathbf{X})$. La proposition suivante découle directement de la définition de AP_i :

Proposition V.2.1 *La contrainte c est 2B–consistante sur $\mathbf{X}$ ssi pour tout i dans $\{1, \ldots, k\}$, $\mathbf{x_i} = AP_i(c, \mathbf{X})$.*

En général, l'approximation AP_i ne peut pas être calculée efficacement car il est difficile de définir les fonctions `min` et `max` quand c n'est pas monotone[5]. En particulier, quand la variable x a des occurrences multiples dans c, déterminer les fonctions `min` et `max` nécessite d'isoler x. Comme cette transformation symbolique n'est pas toujours possible, le problème est résolu en décomposant le système de contraintes en un ensemble de contraintes primitives pour lesquelles AP_i peut être

[5]B. Faltings [16] a introduit une méthode pour calculer l'approximation sans définir la fonction de projection. Cependant cela nécessite une analyse complexe pour trouver les extrema.

calculé facilement [33]. Les contraintes primitives sont générées de façon purement syntaxique en introduisant de nouvelles variables[6].

Définition V.2.3 (Décomposition du système de contraintes)
Soit $P = (X, C)$ un CSP et $c \in C$ une contrainte. Soit $\mathcal{M}_c \subseteq X$ l'ensemble des variables qui ont des occurrences multiples dans c.

- *$decomp(c)$ est l'ensemble de contraintes obtenu en substituant dans c chaque occurrence des variables $x \in \mathcal{M}_c$ avec une nouvelle variable y de domaine $\mathbf{y} = \mathbf{x}$ et en ajoutant la contrainte $x = y$*[7].
- *$new_{(x,c)}$ est l'ensemble des variables introduites pour supprimer les occurrences multiples de la variable $x \in \mathcal{M}_c$ dans c*
- *$X_{new} = \bigcup\{new_{(x,c)} \mid x \in X$ et $c \in C\}$.*
- *P_{decomp} est le CSP (X', C') où $X' = X \cup X_{new}$, et $C' = \{decomp(c) \mid c \in C\}$.*

L'inconvénient de cette décomposition est qu'une consistance *locale* comme l'arc–consistance donne un filtrage plus faible sur le système décomposé que sur le système initial. En effet, la décomposition disperse sur plusieurs contraintes les relations entre les variables d'une même contrainte. Comme les consistances locales ne travaillent que sur une contrainte à la fois, les relations de la contrainte initiale ne sont pas prises en compte. Ceci est illustré sur l'exemple suivant.

Exemple V.4 (Décomposition du système de contraintes) Soit $c : x_1 + x_2 - x_1 = 0$ une contrainte et $\mathbf{x_1} = [-1, 1]$, $\mathbf{x_2} = [0, 1]$ les domaines de x_1 et x_2. Puisque x_1 apparaît deux fois dans c, sa deuxième occurrence est remplacée par une nouvelle variable x_3 : $decomp(c) = \{x_1 + x_2 - x_3 = 0, x_1 = x_3\}$.
Dans ce nouveau système de contraintes, chaque projection peut être facilement calculée grâce à l'arithmétique des intervalles. Par exemple, $AP_1(x_1 + x_2 - x_3 = 0, (\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}))$ est $\mathbf{x_1} \cap (\mathbf{x_3} \ominus \mathbf{x_2})$[8].
Cependant, cette décomposition augmente le problème de localité : la première contrainte est vérifiée indépendamment de la seconde et par conséquent lors du filtrage par 2B–consistance de la contrainte $x_1 + x_2 - x_3 = 0$, x_1 et x_3 peuvent prendre des valeurs distinctes[9]. Plus précisément, la contrainte initiale c n'est pas 2B–consistante puisqu'il n'y a aucune valeur de x_1 qui satisfait c quand $x_2 = 1$. Par contre, le système décomposé $decomp(c)$ est 2B–consistant puisque les valeurs $x_1 = -1$ et $x_3 = 0$ satisfont $x_1 + x_2 - x_3 = 0$ quand $x_2 = 1$. En utilisant la contrainte initiale, la 2B–consistance réduit $\mathbf{x_2}$ à $[0, 0]$ alors que les domaines sont inchangés pour le système décomposé. ♯

[6] À l'époque de ces travaux, nous avons considéré le cas simple de la décomposition en contraintes primitives. D'autres transformations plus efficaces permettent de calculer les fonctions de projection, par exemple pour les fonctions monotones [1].

[7] Si c ne contient pas d'occurrences multiples, alors $decomp(c) = c$.

[8] L'intersection avec le domaine initial $\mathbf{x_1}$ permet d'assurer la propriété de contractance de l'opération de filtrage.

[9] Une consistance globale éviterait ce problème en tenant compte également de la contrainte $x_1 = x_3$.

Box–consistance

La box–consistance [2, 25] est une relaxation de l'arc–consistance plus lâche que la 2B–consistance. Elle revient à remplacer dans la définition de la 2B–consistance la contrainte sur les réels par son extension optimale aux intervalles : chaque variable quantifiée de façon existentielle est remplacée par son intervalle associé, excepté pour la variable dont on teste la 2B–consistance, qui elle est remplacée par ses bornes (i.e. $[\underline{x_i}, \underline{x_i}^+[$ et $]\overline{x_i}^-, \overline{x_i}]$). Ainsi, la box–consistance produit un système de fonctions d'intervalles uni-variables qui peut être résolu par des méthodes numériques telles que Newton. Contrairement à la 2B–consistance, la box–consistance n'exige pas de décomposition du système de contraintes et n'amplifie donc pas le problème de localité.

Définition V.2.4 (Box–consistance)
Soit (X, C) un CSP et $c \in C$ une contrainte k-aire sur les variables $(x_1, \ldots, x_k)$. c est box–consistante ssi pour tout x_i :
$\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}}^+[, \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$,
$\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}},]\overline{\mathbf{x_i}}^-, \overline{\mathbf{x_i}}], \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$.
La fermeture par box–consistance de P est notée $\Phi_{Box}(P)$.

Calcul de la box–consistance

L'algorithme de filtrage par box–consistance proposé dans [2, 23, 24] suit le même schéma itératif que celui du filtrage par arc–consistance (voir section V.1 figure III.2), mais utilise comme brique de base une extension aux intervalles de la méthode de Newton.

Grossièrement, le processus de filtrage consiste à trouver le zéro le plus à gauche et le plus à droite de l'extension naturelle aux intervalles de la contrainte. Tout d'abord, l'intervalle est filtré avec une version sur intervalles de la méthode classique de Newton. Comme la réduction peut ne pas être suffisante pour assurer la box–consistance, une étape de dichotomie est utilisée itérativement pour assurer que la borne gauche est effectivement un zéro. Ce processus de découpage permet aussi d'éviter le problème de la recherche d'une boîte de départ sûre pour la méthode de Newton (voir [25]). De plus, comme mentionné dans [24], même si $\mathbf{f}$ n'est pas dérivable en x, ce processus permet de trouver le zéro le plus à gauche grâce à la dichotomie (alors que la méthode de Newton sur intervalles ne s'applique pas et donc ne réduit pas l'intervalle).

Les sous-sections suivantes définissent la 3B–consistance et la bound–consistance qui sont des généralisations d'ordre supérieur des 2B–consistance et box–consistance. Ce principe est similaire à la généralisation de l'arc–consistance par la consistance de chemin [34][10]. Il s'agit de minimiser l'effet de localité des consistances en faisant intervenir plusieurs variables.

[10] La consistance de chemin considère des couples de variables plutôt qu'une seule variable. Elle a été définie initialement pour les contraintes *binaires* (i.e. faisant intervenir deux variables) [14]. Une contrainte binaire c telle que $Var(c) = \{x_i, x_j\}$ est consistante de chemin pour la variable x_k si pour tous les couples de valeurs (v_i, v_j) qui satisfont c il existe un couple (v_i, v_k) qui satisfait les contraintes c_{ik} telles que $Var(c_{ik}) = \{x_i, x_k\}$ et un couple (v_k, v_j) qui satisfait les contraintes c_{kj} telles que $Var(c_{kj}) = \{x_j, x_k\}$.

3B–consistance

La 3B–consistance consiste à vérifier que le système dans lequel le domaine d'une des variables a été remplacé par ses bornes inférieures et supérieures est 2B–consistant.

Définition V.2.5 (3B–consistance [32])
Soit $P = (X, C)$ un CSP et x une variable de X. Soient :
- $P_{\mathbf{x} \leftarrow [\underline{x}, \underline{x}^+[}$ *le CSP dérivé de P en substituant* $\mathbf{x}$ *avec* $[\underline{x}, \underline{x}^+[$;
- $P_{\mathbf{x} \leftarrow]\overline{x}^-, \overline{x}]}$ *le CSP dérivé de P en substituant* $\mathbf{x}$ *avec* $]\overline{x}^-, \overline{x}]$.

$\mathbf{x}$ *est* **3B–consistant** *ssi* $\Phi_{2B}(P_{\mathbf{x} \leftarrow [\underline{x}, \underline{x}^+[}) \neq P_\emptyset$ *et* $\Phi_{2B}(P_{\mathbf{x} \leftarrow]\overline{x}^-, \overline{x}]}) \neq P_\emptyset$.
Un CSP est 3B–consistant ssi tous ses domaines sont 3B–consistants.

Bound–consistance

La bound–consistance applique le principe de la 3B–consistance à la box–consistance.

Définition V.2.6 (Bound–consistance [23])
Soit (X, C) un CSP et $c \in C$ une contrainte k-aire sur les variables $(x_1, \ldots, x_k)$. c est bound–consistant ssi pour tout x_i :

$$\Phi_{Box}(\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+[, \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})) \neq P_\emptyset,$$
$$\Phi_{Box}(\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}},]\overline{\mathbf{x}}^-, \overline{\mathbf{x}}], \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})) \neq P_\emptyset.$$

Bien que formulée de façon différente, cette définition est similaire à la définition V.2.5 puisque le CSP est bound–consistant si et seulement si toutes les contraintes le sont, c'est-à-dire que les CSP $P_{\mathbf{x} \leftarrow [\underline{x}, \underline{x}^+[}$ et $P_{\mathbf{x} \leftarrow]\overline{x}^-, \overline{x}]}$ sont box–consistants.

V.2.2 Comparaison des consistances

La 2B–consistance et la box–consistance ont pour point commun de définir une propriété sur les bornes des intervalles. La 2B–consistance porte sur la boîte englobante de la fonction, tandis que la box–consistance porte sur l'extension aux intervalles de la fonction. La propriété V.1.1 énonce qu'en cas d'occurrences multiples, la boîte englobante de la fonction est incluse (strictement dans la plupart des cas) dans son extension aux intervalles. Il en découle donc les propriétés suivantes (les preuves peuvent être trouvées dans l'article support section V.5.1) :

1. $\Phi_{2B}(\mathrm{P}) \preceq \Phi_{Box}(\mathrm{P})$. Le filtrage par 2B–consistance est plus fort (i.e. donne un intervalle plus petit) que celui par box–consistance et ils sont égaux quand toutes les variables ont des occurrences uniques.
2. $\Phi_{Box}(\mathrm{P}) \preceq \Phi_{2B}(P_{decomp})$. Puisque la 2B–consistance est une consistance locale, le filtrage par 2B–consistance du système décomposé (nécessaire pour calculer la 2B–consistance) est plus faible que le filtrage par box–consistance sur le système initial.

3. $\Phi_{3B}(P_{decomp}) \preceq \Phi_{Box}(\mathrm{P})$. La 3B–consistance étant moins locale que la 2B–consistance et la box–consistance, le filtrage par 3B–consistance sur le système décomposé est plus fort que la box–consistance sur le système initial.
4. Enfin, nous avons identifié deux exemples qui montrent qu'il n'y a pas de lien entre le filtrage par 3B–consistance sur le système décomposé et le filtrage par 2B–consistance sur le système initial.

L'exemple V.5 montre un système qui est 2B–consistant après décomposition mais n'est pas box–consistant pour illustrer la propriété 2.

Exemple V.5 Soit c la contrainte $x_1 + x_2 - x_1 - x_1 = 0$ où $\mathbf{x_1} = [-1, 1]$ et $\mathbf{x_2} = [0.5, 1]$. c n'est pas box–consistant puisque $[-1, -1^+] \oplus [0.5, 1] \ominus [-1, -1^+] \ominus [-1, -1^+] \cap [0, 0]$ est vide. En effet, d'après la définition des extensions optimales aux intervalles de $+$ et $-$ données dans l'exemple V.1, $[-1, -1^+] \oplus [0.5, 1] \ominus [-1, -1^+] \ominus [-1, -1^+] = [1.5, 2]$. Par contre, *decomp*$(c)$ est 2B–consistant pour $\mathbf{x_1}$ et $\mathbf{x_2}$. ♯

Les deux exemples suivants illustrent la propriété 4.

Exemple V.6 Soit P un CSP défini par $C = \{x_1 + x_2 = 10, x_1 + x_1 - 2x_2 = 0\}$ où $\mathbf{x_1} = \mathbf{x_2} = [-10, 10]$. *decomp*$(x_1 + x_1 - 2x_2 = 0) = \{x_1 + x_3 - 2x_2 = 0, x_3 = x_1\}$. P est 2B–consistant mais P_{decomp} n'est pas 3B–consistant. En effet, les contraintes de P_{decomp} sont $\{c_1 : x_1 + x_2 = 10, c_2 : x_1 + x_3 - 2x_2 = 0, c_3 : x_3 = x_1\}$. Quand x_1 est fixé à 10, c_1 impose que $x_2 = 0$ tandis que c_2 et c_3 imposent que $x_2 = 10$. Par conséquent, $\Phi_{2B}(P_{\mathbf{x_1} \leftarrow [10^-, 10]}) = \emptyset$. Dans ce cas, le lien entre x_1 et x_3 a été préservé par la 3B–consistance qui a fixé la valeur d'une des variables à occurrences multiples. Notons que les étapes itérées de filtrage par 3B–consistance et recherche dichotomique réduisent $\mathbf{x_1}$ et $\mathbf{x_2}$ à $[5, 5]$[11]. ♯

Exemple V.7 Soit c la contrainte $x_1 \times x_2 - x_1 + x_3 - x_1 + x_1 = 0$ où $\mathbf{x_1} = [-4, 3], \mathbf{x_2} = [1, 2]$ et $\mathbf{x_3} = [-1, 5]$. *decomp*$(c) = \{x_1 \times x_2 - x_4 + x_3 - x_5 + x_6 = 0, x_1 = x_4 = x_5 = x_6\}$. c n'est pas 2B–consistante puisqu'il n'y a pas de valeurs dans $\mathbf{x_1}$ et $\mathbf{x_2}$ qui vérifient c quand $x_3 = 5$ (c est en fait la relation $x_1 \times (x_2 - 1) + x_3 = 0$). Cependant, *decomp*$(c)$ est 3B–consistant, puisque la perte de lien entre les deux occurrences de x_1 empêchent le filtrage de x_3 (i.e. on peut trouver des valeurs pour x_4, x_5 et x_6 qui satisfont *decomp*(c) quand x_3 est fixé à -1 et à 5). ♯

V.2.3 Conclusion

Ces travaux ont établi une comparaison fine des consistances partielles utilisées dans les solveurs sur domaines finis. En pratique, les techniques basées sur la box–consistance fournissent un filtrage plus fin que celles basées sur la 2B–consistance, à cause de la décomposition du système de contraintes nécessaire pour calculer les fonctions de projection. Toutefois, en terme d'efficacité, appliquer la méthode de Newton

[11] Un intervalle qui n'est pas 2B–consistant est coupé en deux et le filtrage par 2B–consistance est appliqué à sa partie gauche et droite. À la fin de ce processus itératif, la borne la plus à gauche et la plus à droite qui est 3B–consistante a été trouvée et fournit l'intervalle 3B–consistant.

étendues aux intervalles sur des fonctions uni-variables peut être plus coûteux que de calculer les fonctions de projection sur les contraintes élémentaires. Ces méthodes peuvent donc être utilisées de façon complémentaire : un filtrage plus grossier mais plus efficace avec la 2B–consistance peut éliminer rapidement une partie du domaine qui sera affiné avec la box–consistance.

La section suivante présente un emploi original de la box–consistance.

V.3 Extension de domaines consistants

Cette section présente un algorithme pour étendre le domaine d'une variable dans un CSP qui est défini par un ensemble de contraintes non linéaires *sur les réels*, et dont les domaines contiennent uniquement des points solution.

V.3.1 Motivations et état de l'art

Motivations

Cette étude a été motivée par un problème pratique qui se pose dans une grande classe de problèmes en électro-mécanique et applications de génie civil. Ces problèmes sont souvent sous–contraints, et possèdent donc un grand nombre de solutions. Le concepteur veut donc connaître un sous-ensemble de ces solutions, et pouvoir "l'élargir" tout en s'assurant que le nouvel ensemble construit est toujours solution du système. L'objectif est d'offrir le plus grand degré de liberté pour une variable ce qui permet en pratique d'agrandir la tolérance du composant associé, et donc d'abaisser son coût.

Pour ce type de problèmes, les méthodes de 2B(3B)–consistance et box(bound)–consistance étudiées à la section précédente ne sont pas bien adaptées puisqu'elles calculent un sur-ensemble des solutions mais ne peuvent pas assurer qu'il existe une solution à l'intérieur des intervalles (arbitrairement petits) calculés. Nous avons donc défini la notion de *i–consistance* qui représente une boîte qui contient uniquement des tuples solution (i.e. pour toute contrainte c et pour toute valeur $\tilde{X}$ de la boîte i–consistante, la relation $c(\tilde{X})$ est satisfaite). Le problème revient alors à partir d'une boîte i–consistante et à l'agrandir selon l'axe d'une des variables du système, tout en préservant la i–consistance.

Etat de l'art

Les travaux connexes existant à l'époque ne proposaient pas d'implémentation réellement efficace. En effet, deux approches principales existaient. Sam-Haroud et Faltings [43] avaient proposé une approche basée sur une méthode classique en synthèse d'image nommée 2^k arbres. L'idée est de classifier les parties de l'espace en trois catégories : les parties noires ne contiennent aucune solution, les parties grises contiennent des solutions, mais contiennent également des points qui ne sont pas solution, et les parties blanches contiennent uniquement des points qui sont solution[12]. Les boîtes blanches correspondent à la définition de la i–consistance. Afin

[12]La couleur des boîtes est déterminée grâce à des méthodes d'analyse numérique qui peuvent être coûteuses.

d'affiner le calcul de l'espace des solutions, les parties grises sont découpées en parties plus petites qui sont de nouveau classifiées en parties noires, blanches et grises. Le processus de décomposition s'arrête quand la taille des parties devient plus petite qu'une valeur donnée. Cette méthode permet aussi bien de calculer un sous-ensemble de l'espace solution (i.e. union des boîtes blanches) que d'étendre le domaine d'une variable d'un CSP consistant. Cependant, la complexité en espace et en temps est exponentielle.

D'autre part, Ward et al. [45] avec la notion "d'intervalles étiquetés", ont proposé un cadre formel qui inclut la notion de boîte intérieure nécessaire à l'extension d'un sous-ensemble de solutions. En effet, les intervalles sont étiquetés avec quatre étiquettes possibles : "only", "every", "some" et "none". Si $\mathbf{x}$ est étiqueté par "only", alors les tuples de solution prennent uniquement leurs valeurs pour x dans $\mathbf{x}$. Si $\mathbf{x}$ est étiqueté par "every", alors chaque valeur de x dans $\mathbf{x}$ appartient à un tuple solution. Si $\mathbf{x}$ est étiqueté par "some", alors il existe au moins un tuple solution tel que x prend sa valeur dans $\mathbf{x}$. Enfin, si $\mathbf{x}$ est étiqueté par "none", alors il n'existe aucun tuple solution tel que la valeur de x est dans $\mathbf{x}$. Un système pour lequel chaque variable est étiquetée par "every" correspond à notre notion de i–consistance. Cependant, les règles d'inférence proposées par Ward et al. [45] qui permettent d'effectuer une propagation pour les intervalles étiquetés, ne considèrent pas le cas où deux variables sont étiquetées par "every". De plus, ces règles d'inférence supposent une propriéte de forte monotonie et de continuité sur le système de contraintes.

Contributions

Nous avons donc proposé une approche plus efficace mais moins générale, qui nécessite de connaitre un sous-ensemble initial des solutions. De plus, pour que l'extension d'une solution existante ait un sens, nous supposons que le système de contraintes contient uniquement des inégalités[13].

Pour étendre le domaine d'une variable par la droite (i.e. par sa borne supérieure), nous procédons de la façon suivante[14] :

1. Rechercher un sous-ensemble de l'espace des solutions ; cet espace peut être réduit à un unique point.
2. Sélectionner la variable dont le domaine doit être prolongé.
3. Définir pour chaque inégalité une fonction d'extremum.
4. Rechercher la plus petite solution (i.e. la plus à gauche) de toutes les fonctions d'extremum.

L'étape 1 peut être résolue par des méthodes d'analyse numérique, ou bien le sous-ensemble initial des solutions peut être fourni par l'utilisateur. L'étape 2 est liée au problème : on peut choisir par exemple la variable qui modélise le composant dont le coût est le plus critique. L'étape 3 fait appel à la définition des fonctions

[13] Si le système contient des égalités, cela peut impliquer que certains domaines de la solution initiale soient réduits à une seule valeur, ce qui rend impossible l'extension pour cette variable. Cela ne pose pas de problème en soi mais aurait compliqué la formalisation du problème.

[14] Ici, comme dans le reste de la section, nous présenterons uniquement la façon d'étendre une variable par la droite, puisque l'extension à gauche est symétrique.

de projection. Enfin, la recherche de solution de l'étape 4 est mise en œuvre de deux façons. Si les contraintes sont primitives, alors nous calculons les fonctions de projection associées. Sinon, la borne maximale est calculée de façon efficace (mais pas toujours optimale) par la box–consistance qui produit la plus petite boîte extérieure qui contient les fonctions d'extremum.

Nous introduisons tout d'abord des définitions, en particulier la notion de boîte intérieure. Nous montrons ensuite comment agrandir le domaine d'une variable.

V.3.2 Définitions

Définition V.3.1 (boîte)
Une (k–)boîte $I_1 \times \ldots \times I_k$ est la partie de l'espace k–dimensionnel définie par le produit cartésien des intervalles $(I_1, \ldots, I_k)$.

e–consistance et i–consistance

Nous introduisons deux nouvelles approximations de l'arc–consistance : l'e–consistance (e pour e-xtérieur) et l'i–consistance (i pour i-ntérieur).

Définition V.3.2 (e–consistance)
Soit $P = (X, C)$ un CSP arc–consistant. Un CSP $P' = (X', C)$ est **e–consistant** *ssi pour tout i $\mathbf{x}'_i = \square x_i$.*

En d'autres termes, un CSP $P = (X, C)$ est e–consistant si et seulement si $P' = (X', C)$ est arc–consistant et X correspond à la plus petite boîte qui contient toutes les valeurs de X'.

Définition V.3.3 (i–consistance)
Soit $P = (X, C)$ un CSP. P est **i–consistant** *ssi $\forall c \in C, \forall \tilde{X} \in \mathbf{X} \ : \ c(\tilde{X})$.*

La différence fondamentale entre l'i–consistance et les consistances partielles étudiées à la section précédente est que ces dernières définissent des régions qui contiennent toutes les solutions ainsi qu'éventuellement des tuples qui ne sont pas solution tandis que la i–consistance définit une région qui ne contient que des solutions. La figure V.2 illustre les notions d'e–consistance et i–consistance.

Extension i–consistante à droite d'une variable

L'extension à droite d'une variable est définie de la façon suivante :

Définition V.3.4 (Extension i–consistante à droite de x)
Soit $P = (X, C)$ un CSP i–consistant. $P' = (X', C)$ est une **extension i–consistante à droite** *de x pour P ssi :*

1. $\forall \mathbf{x}_i \in X \setminus \{\mathbf{x}\}, \ \mathbf{x}_i = \mathbf{x}'_i$
2. $\mathbf{x} \subset \mathbf{x}'$
3. $\underline{\mathbf{x}'} = \underline{\mathbf{x}}$
4. *P' est i–consistant*

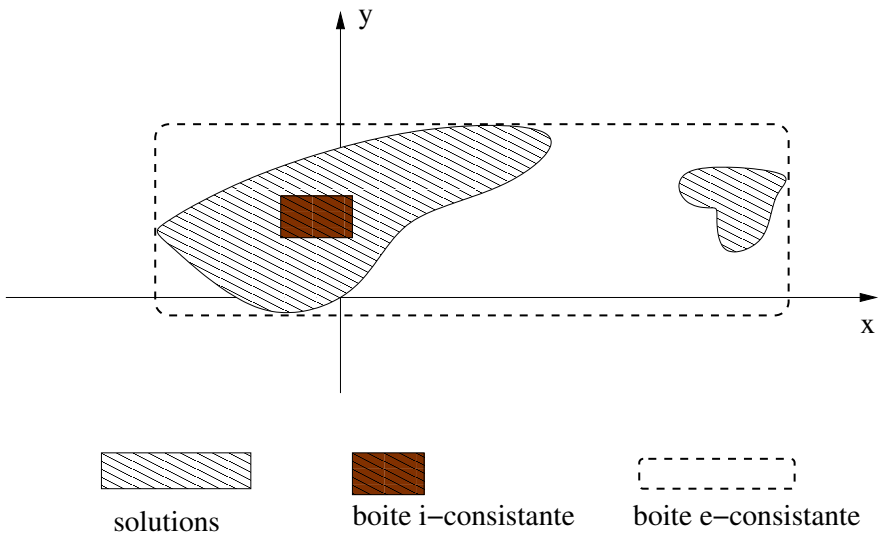


FIG. V.2 – Exemple de boîtes i–consistante et e-consistante

Une extension i–consistante à droite de x est **maximale** *si elle est la plus grande au sens de l'inclusion.*

Le point 1 énonce que les domaines des variables différentes de x sont inchangés. Le point 2 que X' est une extension de X au sens de l'inclusion. Le point 3 que la borne gauche n'a pas changé (il s'agit d'une extension à droite). Et enfin le point 4 assure que le CSP obtenu est toujours i–consistant.

L'exemple suivant, illustré sur la figure V.3 montre les extensions maximales à droite et à gauche d'un CSP dont l'espace des solutions est délimité par deux paraboles.

Exemple V.8 (Extension d'une solution à droite et à gauche pour x)
Soit P le CSP défini par $C = \{x^2 - y \leq 0, -x^2 + 20 \leq y\}$, avec pour domaines associés $\mathbf{x} = [-2, 2]$ et $\mathbf{y} = [6, 12]$. Ce système est i–consistant :
$\forall(\tilde{x}, \tilde{y}) \in (\mathbf{x}, \mathbf{y})\ \tilde{x}^2 - \tilde{y} \leq 0 \wedge -\tilde{x}^2 + 20 \leq \tilde{y}$
Le problème qui nous intéresse ici est de trouver le plus grand intervalle $\mathbf{x}'$ avec $\mathbf{x} \subseteq \mathbf{x}'$ tel que le CSP P' obtenu à partir de P en remplaçant $\mathbf{x}$ par $\mathbf{x}'$ soit i–consistant.
La figure V.3 montre la boîte originale (rosée) et l'extension de x (parties hachurées à gauche et à droite). Notons que nous pourrions également effectuer une extension i–consistante de y à $[6, 14]$ à partir de la boîte obtenue par extension de x. Mais cette extension de y est beaucoup plus petite que celle que nous aurions obtenue en prolongeant directement la boîte initiale sur y (Y aurait été prolongé à $[4, 16]$). De façon générale, le résultat des extensions successives par i–consistance de plusieurs variables dépend de l'ordre de traitement de ces variables. ♯

Nous définissons maintenant les fonctions d'extremum qui vont permettre de calculer la valeur limite à laquelle une variable peut être étendue.

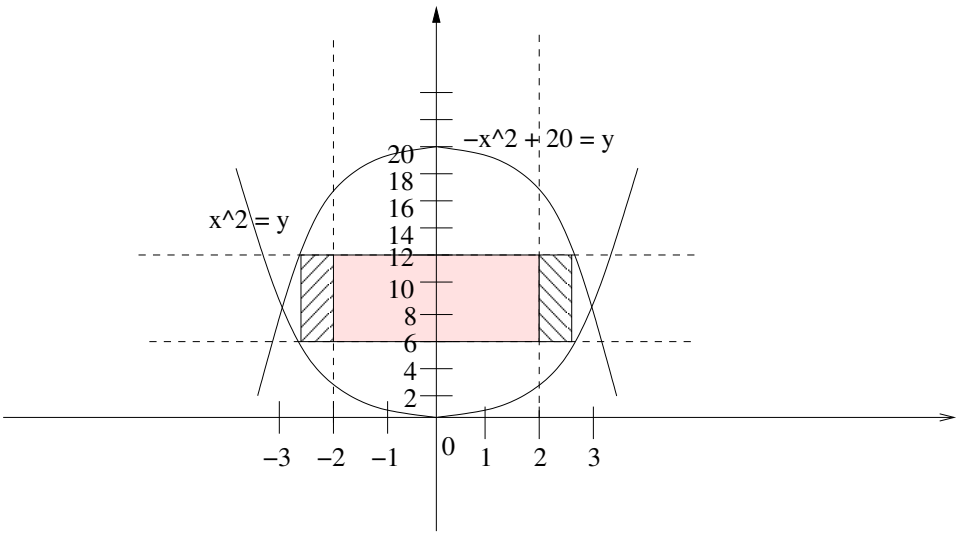


FIG. V.3 – Extension i–consistante maximale sur la variable x

Fonctions d'extremum

Soit c une inégalité, nous notons c_{equ} l'équation qui correspond à c. Plus précisément, si c est définie par une expression de la forme $f(x_1, \ldots, x_n) \leq 0$ ou $f(x_1, \ldots, x_n) \geq 0$, alors c_{equ} est l'équation $f(x_1, \ldots, x_n) = 0$.

Définition V.3.5 (Fonction extremum optimale)
Soit $P = (X, C)$ un CSP, x une variable de P et c une contrainte de C. La **fonction extremum optimale** *de la contrainte c pour la variable x est $F_c^{min(x)}(X) = min(\pi_x(\tilde{X}) \mid c_{equ}(\tilde{X}))$.*

En d'autres termes, $F_c^{min(x)}(X)$ est une fonction extremum optimale de c pour la variable x si $F_c^{min(x)}(X)$ calcule la plus petite valeur de x qui est solution de c_{equ} dans l'espace délimité par X. Par convention, $F_c^{min(x)}(X)$ renvoie $\overline{x}$ quand c_{equ} n'a pas de solution dans l'espace délimité par X.

Définition V.3.6 (Approximation des fonctions d'extremum)
Soit $P = (X, C)$ un CSP, c une contrainte de C et $F_c^{min(x)}(X)$ une fonction extremum optimale de c_{equ} pour la variable x. $AF_c^{min(x)}(X)$ est une approximation sûre de $F_c^{min(x)}(X)$ ssi : $AF_c^{min(x)}(X) < F_c^{min(x)}(X)$

L'approximation est sûre dans le sens qu'elle est inférieure à l'extremum optimal. Par conséquent, lors de l'extension du domaine à droite, on est sûr de ne pas dépasser cet extremum.

V.3.3 Calcul d'une extension i–consistante à droite

Principe

Pour définir de façon opérationnelle l'extension maximale i–consistante à droite de x pour un CSP $P = (X, C)$, nous introduisons le domaine $\mathbf{x}_{max} = [\overline{\mathbf{x}}, +\infty[$ qui est le domaine $\mathbf{x}$ étendu à l'infini, ainsi que le domaine $\mathbf{X}_{max} = X_{\mathbf{x} \leftarrow [\overline{\mathbf{x}}, +\infty[}$ qui est le domaine initial $\mathbf{X}$ où $\mathbf{x}$ est remplacé par $\mathbf{x}_{max}$.

Par définition, la fonction extremum optimale $F_c^{min(x)}(\mathbf{X}_{max})$ donne la valeur la plus à gauche dans $\mathbf{x}_{max}$ qui satisfait l'égalité c_{equ}, puisqu'elle est le minimum des projections sur $\mathbf{x}_{max}$ des solutions de c_{equ}. Comme nous partons d'un intervalle $\mathbf{x}$ qui est déjà solution et qui par définition est à gauche de l'espace délimité par c_{equ}, nous savons que $F_c^{min(x)}(\mathbf{X}_{max})$ est la borne maximale possible pour x quand on considère seulement la contrainte c. Pour plusieurs contraintes, il suffit donc de prendre le minimum de toutes les fonctions extremum optimales associées aux contraintes qui font intervenir la variable x. Nous obtenons donc la proposition suivante (la preuve est détaillée dans l'article support section V.5.2) :

Proposition V.3.1 *Soit $P = (X, C)$ un CSP i–consistant. Soit $P' = (X', C)$ tel que :*

- $\forall \mathbf{x}_i \in X \setminus \{\mathbf{x}\} : \mathbf{x}_i = \mathbf{x}'_i$
- $\mathbf{x} = [\underline{\mathbf{x}}, min_{c \in C}(F_c^{min(x)}(\mathbf{X}_{max}))]$

Alors P' est une extension i–consistante à droite sur x pour P.
De plus, P' est une extension **maximale** *i–consistante à droite sur x pour P.*

La propriété V.3.1 est illustrée figure V.4 et dans l'exemple V.9.

Exemple V.9 (Utilisation de $\mathbf{x}_{max}$) Soit P le CSP défini par $C = \{x^2 - y \leq 0, -2x + 10 \leq y\}$, avec pour domaines associés $\mathbf{x} = [-1, 1]$ et $\mathbf{y} = [2, 6]$. Ce système est i–consistant. Pour étendre le domaine de la variable x à droite, il suffit de calculer le minimum des extremums des courbes $x^2 - y = 0$ et $-2x + 10 = y$ intersectées avec le domaine $\mathbf{x}_{max} = [1, +\infty[$. Ceci est illlustré figure V.4 où la boîte i-consistante initiale est rosée, $\mathbf{x}_{max}$ est représenté par un rectangle de briques, et les extremums des courbes par des points. Dans ce cas, l'extension maximale est obtenue pour la valeur $\overline{\mathbf{x}} = \sqrt{2}$; elle correspond à l'intersection de la droite $y = 2$ avec la courbe $x^2 - y = 0$. ♯

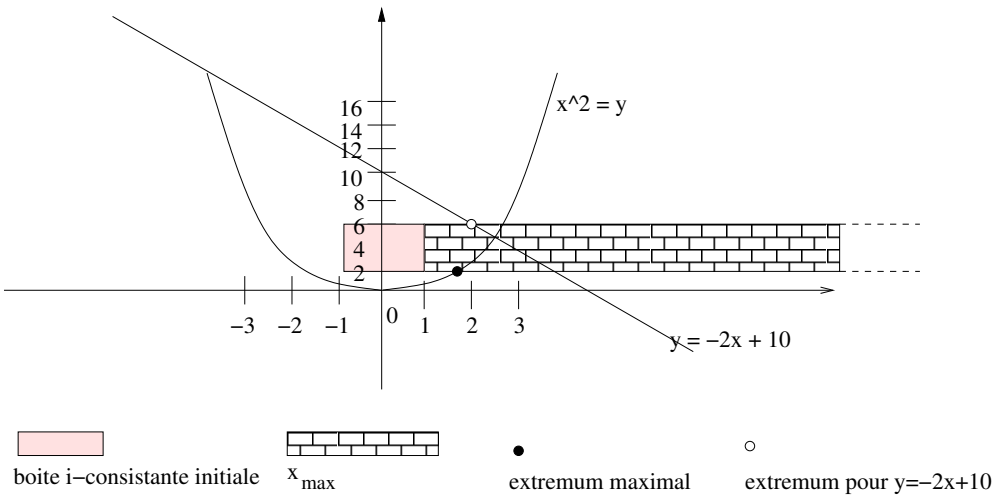


FIG. V.4 – Utilisation de $\mathbf{x}_{max}$ pour le calcul de l'extension à droite

Pour calculer l'extension optimale à droite de la variable x d'un système de contraintes contenant les contraintes C, il faut donc calculer le minimum de toutes les fonctions d'extremum dans l'espace délimité par $\mathbf{X}_{max}$. L'algorithme associé est linéaire si les fonctions d'extremum peuvent être calculées en temps constant.

Calcul des fonctions d'extremum

La fonction extremum optimale pour la variable x de la contrainte c peut être calculée aisément si c est primitive. Pour une contrainte c *non primitive*, nous ne calculons pas la fonction extremum optimale mais nous utilisons une approximation de la boîte e–consistante pour c_{equ} dans l'espace délimité par le domaine $\mathbf{X}_{max}$, c'est-à-dire une approximation de $F_c^{min(x)}(\mathbf{X}_{max})$. Plus précisément, soit un CSP $P = (X, C)$ i–consistant et une inégalité $c \in C$. Pour calculer une approximation sûre de la fonction d'extremum pour x de la contrainte c, nous calculons un intervalle box–consistant pour x par rapport à c_{equ}. En effet, la box–consistance produit un intervalle $\mathbf{x}'$ tel que $\Pi_x(sol(c_{equ}, \mathbf{X}_{max})) \subset \mathbf{x}'$ où $sol(c_{equ}, \mathbf{X}_{max})$ est l'ensemble des solutions de c_{equ} dans le domaine délimité par $\mathbf{X}_{max}$. Par conséquent, $\underline{\mathbf{x}'} = AF_c^{min(x)}(\mathbf{X}_{max})$. Les méthodes de calcul de la box–consistance fournissent une façon efficace de calculer cette approximation. Le point clef est que les fonctions d'extremum sont des fonctions uni-variables qui peuvent être résolues par la méthode de Newton.

L'utilisation de la box–consistance pour approximer les fonctions d'extremum est illustrée dans la figure V.5 qui reprend le même système de contraintes que l'exemple V.9.

La méthode présentée dans cette section a été appliquée à plusieurs exemples dont un exemple de balistique. Le lecteur intéressé peut se reporter à l'article support V.5.2 page 248.

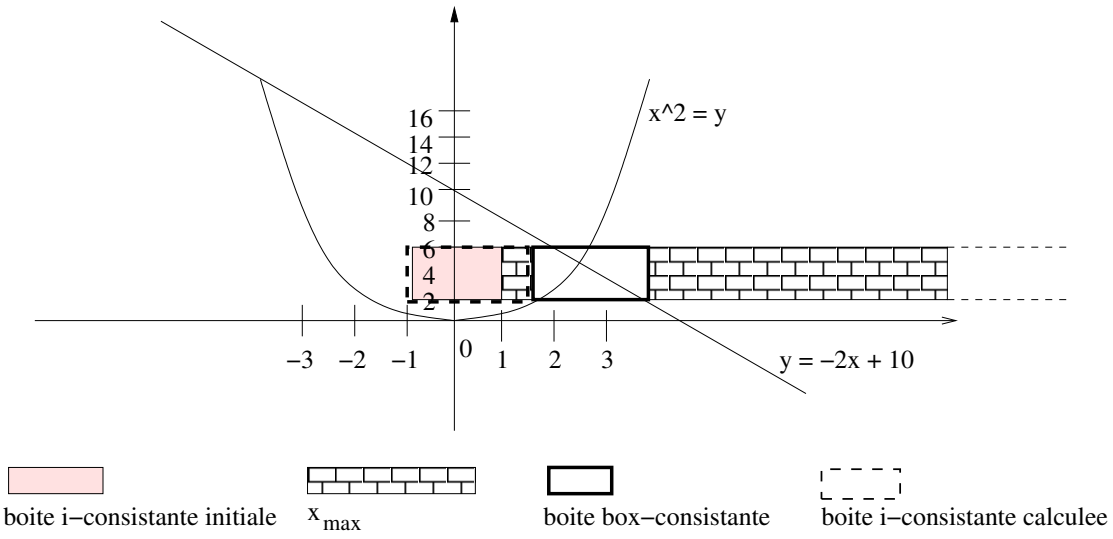


FIG. V.5 – Utilisation de la box–consistance pour le calcul de l'extension à droite

V.4 Discussion

Dans cette section, je décris tout d'abord deux travaux marquants pour résoudre le problème de la boîte intérieure de la section V.3. Je présente ensuite différents solveurs pour résoudre les CSP continus, en me concentrant sur les travaux connexes à ceux de mon équipe. Cela me permet aussi d'ouvrir une discussion sur les différents types de solveurs, et le choix d'un tel solveur dans le cadre de la vérification formelle des programmes.

V.4.1 Extension des consistances intérieures

Une méthode de calcul d'une boîte intérieure similaire à la méthode présentée section V.3 a été proposée dans [3]. Le problème est plus général puisqu'il s'agit de calculer une boîte intérieure alors que nos travaux consistaient à élargir une telle boîte. Le calcul d'une approximation intérieure de relations sur les réels a tout d'abord été associé à la gestion de quantificateurs universels sur les variables du CSP. En effet, alors que les définitions usuelles des consistances (voir par exemple la 2B–consistance, définition V.2.1 page 215) utilisent un quantificateur *existentiel* pour la variable dont on assure la consistance, un quantificateur *universel* est nécessaire pour la consistance intérieure (voir définition V.3.3 page 223). Ensuite, des algorithmes de résolution de tels CSP ont été proposés. L'idée de base est de calculer une boîte extérieure qui englobe les solutions de la négation des contraintes mises en jeu. Ceci est orthogonal à notre approche : au lieu de calculer la solution la plus à gauche sur le domaine $[\overline{\mathbf{x}}, +\infty[$, on calcule la valeur la plus à droite qui n'est pas solution de la contrainte c'est-à-dire qui est solution de sa négation. Cette approche a été utilisée pour modéliser les mouvements d'une caméra dans [5].

L'extension d'une boîte initialement i–consistante a été récemment formalisée

dans le cadre des intervalles modaux [20]. Les intervalles modaux manipulent des intervalles *dégénérés* dont la borne droite peut être inférieure à la borne gauche. Ils fournissent un cadre intéressant pour modéliser les CSP quantifiés (voir [18] pour la formulation initiale et la thèse d'Alexandre Goldsztejn [19] pour une reformulation largement adoptée). L'approche proposée permet de partir d'une boîte i–consistante initiale réduite à un seul point et d'élargir le domaine d'une variable, avec la restriction que cette variable ne doit pas avoir d'occurrences multiples. Elle combine un résultat de l'analyse des intervalles modaux pour vérifier qu'une boîte est intérieure, à la notion de fonction de projection issue des techniques CSP (qui est redéfinie dans le cadre des intervalles modaux). Le calcul de ces fonctions de projection est efficace, mais est limitée par l'hypothèse d'absence d'occurrences multiples.

Le sous-section suivante examine les solveurs non-linéaires.

V.4.2 Solveurs non linéaires

Mes recherches actuelles sur la vérification de programmes soulèvent inévitablement la question de la vérification de programmes contenant des expressions non linéaires. Je présente donc ici un aperçu des outils que je pourrais utiliser à cette fin. Je me concentre sur les outils qui font appel à la fois aux techniques de satisfaction de contraintes et aux méthodes d'analyse des intervalles. D'autre part, je présente un outil développé dans l'équipe *CeP* écrit dans le cadre du projet $V3F$[15] pour la vérification de programmes en nombres flottants. Ces points seront repris dans les perspectives de la section VI.1.

Solveurs sur les réels

Les familles des *2B* et *box* consistances sont utilisées dans de nombreux solveurs sur les réels. La box–consistance a été implémentée en particulier dans *Numerica* [23], mais ce système n'est plus disponible. La bibliothèque *ALIAS* [35] fournit de nombreuses méthodes d'analyse par intervalles (e.g. basées sur Newton, sur les Jacobien, ...) et offre la possibilité d'utiliser de la *3B–consistance* au lieu de la technique classique de bissection pour rechercher les solutions. *RealPaver* [21] gère des intervalles pour modéliser et résoudre des systèmes de contraintes non linéaires, équations ou contraintes d'inégalité sur les nombres entiers ou réels. Les algorithmes combinent efficacement des méthodes d'analyse d'intervalle (comme la méthode de Newton étendue aux intervalles) et des techniques de satisfaction de contraintes pour réduire les intervalles en utilisant plusieurs consistances (box–consistance, hull–consistance et 3B–consistance). Comme son nom l'indique, *RealPaver* fournit un pavage de l'espace des solutions en gérant des unions d'intervalles ; ceci fournit une représentation des solutions plus fine qu'un simple produit cartésien des intervalles. La bibliothèque *Ibex* [9], est également une bibliothèque qui combine calcul d'intervalles (via la bibliothèque BIAS/PROFIL) et techniques de satisfaction de contrainte, et qui permet de calculer un pavage de l'espace des solutions. Cette bibliothèque implémente le langage *Quimper* pour décrire le système en terme de *contracteurs* [8], qui combinent

[15] Validation et Vérification en présence de calculs à Virgule Flottante, de l'ACI Sécurité Informatique.

description du problème (e.g quelle propriété doit être vérifiée pour que le système soit satisfiable) et manière de le résoudre (e.g. combinaison d'une méthode de Newton et de la propagation). L'algorithme de pavage prend en entrée un ensemble de contracteurs.

D'autres bibliothèques comme *GlobSol* ou *ICOS* [27, 28] intègrent des techniques d'optimisation globale et de relaxations linéaires des contraintes non-linéaires. Le projet *COCONUTS* [39] a permis de comparer ces bibliothèques en terme d'efficacité. La bibliothèque *ICOS* développée dans notre équipe par Yahia Lebbah, fait appel à une technique d'approximation linéaire nommée *Quad* qui s'applique aux contraintes d'égalité et inégalité quadratiques (i.e. contenant le produit de deux variables ou le carré d'une variable) [29]. Plutôt que d'utiliser des consistances locales comme la 2B–consistance ou la box–consistance, l'idée est d'effectuer un filtrage global (e.g par des méthodes de programmation linéaire) sur une relaxation linéaire du système. Cette relaxation transforme l'expression non linéaire en une conjonction d'expressions linéaires qui font intervenir les bornes de l'intervalle. Le système est alors résolu et si les bornes des intervalles ont été réduites, une nouvelle approximation est calculée avec ces nouvelles bornes. Ce processus itératif continue jusqu'à ce qu'il n'y ait plus de réduction des domaines. La comparaison effectuée dans [39] a montré que *ICOS* est particulièrement robuste. Notons que le principe d'approximation linéaire simple (i.e. sans itération) pourrait aussi être utilisé pour décider efficacement du chemin à prendre lors de l'exécution symbolique des programmes (voir perspectives section VI.1.3 page 262).

Toutes les bibliothèques précédentes calculent des intervalles dont les bornes sont des flottants mais qui contiennent l'ensemble des solutions *réelles* du CSP. La sous-section suivante s'intéresse au calcul d'un ensemble de solutions *flottantes*.

Solveurs sur les flottants

La problématique spécifique des CSP sur les flottants a été introduite dans [36]. Il s'agit de trouver les solutions d'un **F***CSP*, un CSP dont les domaines sont des *ensembles finis de nombres flottants*. La définition des $FCSP$ se justifie par la différence fondamentale entre les réels et leur représentation finie en machine avec les flottants. En effet, des relations qui sont vraies sur les réels peuvent être fausses sur les flottants, et de même, une contrainte qui n'a pas de solution sur les réels peut en avoir une sur les flottants. Ce point est particulièrement important dans le cadre de la validation des programmes contenant des nombres flottants. Par exemple, pour analyser la faisabilité d'un chemin fourni par un jeu de test, un mécanisme d'exécution symbolique permet de calculer les conditions du chemin (i.e. ensemble de contraintes sur les variables pour assurer que le chemin est pris). Si l'exécution symbolique se fait en calculant le domaine des variables par un solveur sur les réels, il se peut que la condition du chemin ne soit pas effective sur les flottants (ceci est très bien illustré dans [7]). Même si les $FCSP$ ont des *domaines finis*, les techniques d'énumération de type arc–consistance ne peuvent pas être mises en jeu du fait de la taille des domaines (il y a plus de 10^8 flottants dans $[-1, 1]$ [36]). De plus, l'évaluation d'une contrainte sur les flottants est complexe et dépend de nombreux paramètres comme le mode d'arrondi ou l'ordre d'évaluation des sous-expressions. Par conséquent, une

nouvelle consistance, la *FP–box–consistance*, a été définie ; elle est basée sur la box–consistance. Le solveur FPCS (pour Floating Point Constraint Solver) développé dans notre équipe par Claude Michel dans le cadre du projet $V3F$ met en œuvre l'algorithme de filtrage associé à cette consistance [31].

Un tel solveur sur les flottants sera la brique élémentaire pour vérifier des programmes contenant des variables flottantes, comme expliqué dans les perspectives de mes travaux sur la vérification des programmes (voir perspectives section VI.1.3 page 262).

V.5 Articles supports

V.5.1 Comparing Partial Consistencies

H.Collavizza, F.Delobel, M. Rueher. Comparing Partial Consistencies. Reliable Computing , Kluwer Academic Publishers, Vol.5(3), pp. 213-228, 1999.

Reliable Computing, 5, 1–16 (1999)

Comparing Partial Consistencies *

HÉLÈNE COLLAVIZZA, FRANÇOIS DELOBEL AND MICHEL RUEHER

{collavizza,delobel,rueher}@essi.fr

Université de Nice–Sophia-Antipolis, I3S
ESSI, 930, route des Colles - B.P. 145
06903 Sophia-Antipolis, France

Abstract. Global search algorithms have been widely used in the constraint programming framework to solve constraint systems over continuous domains. This paper precisely states the relations among the different partial consistencies which are main emphasis of these algorithms.
The capability of these partial consistencies to handle the so-called dependency problem is analysed and some efficiency aspects of the filtering algorithms are mentioned.

Keywords: Constraint systems, interval narrowing, partial consistencies

1. Introduction

Global search algorithms that are based upon partial consistency filtering techniques have proven their efficiency to solve non-trivial constraint systems over the reals. For instance, systems like `Newton` and `Numerica` [2, 28] behave better than interval methods on classical benchmarks of numerical analysis and interval analysis (e.g., Moré-Cosnard non-linear integral equation, Broyden banded functions). Moreover, it has been shown recently [9] that combining algorithms based on different partial consistencies can even lead to better performances.

These global search algorithms are actually "branch and prune" algorithms, i.e., algorithms that can be defined as an iteration of two steps :

1. *Pruning the search space* by reducing the intervals associated with the variables;
2. *Generating subproblems* by splitting the domains of a variable (the choice of the variable may be non deterministic or based on some heuristic).

The pruning step achieves a filtering of the domains, in other words, it reduces the intervals associated with the variables until a given partial consistency property is satisfied.

1.1. Partial consistencies

Informally speaking, a constraint system C satisfies a partial consistency property if a relaxation of C is consistent. For instance, local consistency just requires that, taken individually, the constraints are consistent. The relevance of consistency

* This is a revised version of the paper presented at the 4th International Conference on Constraint Programming [6].

properties is that whenever a consistency property is violated, there is an associated recipe for pruning some interval. Most constraint solvers over finite domains [18, 27] are based on a partial consistency named *Arc–Consistency*. Assume c is a k-ary constraint over variables $(x_1, .., x_k)$; c is arc-consistent if, for any value in $\mathbf{x_i}$, there exists at least one value in each domain $\mathbf{x_j}(j \neq i)$ such that c holds. In the same way, many solvers over continuous domains [25, 28, 1] rely upon relaxations of Arc–Consistency. A relaxation of Arc–Consistency has also been used in the context of global optimization [22].

2B–Consistency (also known as hull consistency) [5, 3, 14, 15] is a relaxation of Arc–Consistency which only requires to check the Arc–Consistency property for each bound of the intervals. The key point is that this relaxation is more easily verifiable than Arc–Consistency itself. Informally speaking, variable x is 2B–Consistent for constraint $''f(x, x_1, \ldots, x_n) = 0''$ if the lower (resp. upper) bound of the domain of x is the smallest (resp. largest) solution of $f(x, x_1, \ldots, x_n)$. Box–Consistency [2, 11] is a coarser relaxation (i.e., it allows more stringent pruning) of Arc–Consistency than 2B–Consistency. Variable x is Box–Consistent for constraint $''f(x, x_1, \ldots, x_n) = 0''$ if the bounds of the domain of x correspond to the leftmost and the rightmost zero of the optimal interval extension of $f(x, x_1, \ldots, x_n)$.

3B–Consistency and *Bound-consistency* are higher order extensions of 2B–Consistency and Box–Consistency which have been introduced to limit the effects of a strictly local processing:

- *3B–Consistency* [15] is a relaxation of path consistency [8], a higher order extension of Arc-Consistency. Roughly speaking, 3B–Consistency checks whether 2B–Consistency can be enforced when the domain of a variable is reduced to the value of one of its bounds in the whole system;
- *Bound-consistency* [28, 24] applies the principle of 3B–Consistency to Box–Consistency : Bound-consistency checks whether Box–Consistency can be enforced when the domain of a variable is reduced to the value of one of its bounds in the whole system.

1.2. Aim of the paper

This paper investigates the relations between *2B–Consistency*, *Box–Consistency*, *3B–Consistency* and *Bound-consistency*. More precisely, we prove the following properties:

- 2B–Consistency algorithms actually achieve a weaker filtering (i.e., a filtering that yields bigger intervals) than Box–Consistency, especially when a variable occurs more than once in some constraint (see proposition 6). This is due to the fact that 2B–Consistency algorithms require a decomposition of the constraints with multiple occurrences of the same variable;
- The filtering achieved by Box–Consistency algorithms is weaker than the one computed by 3B–Consistency algorithms (see proposition 8).

This paper also provides an analysis of both the capabilities and the limits of the filtering algorithms which achieve these partial consistencies. We pay special attention to their ability to handle the so-called dependency problem [10].

Layout of the paper

Section 2 reviews some basic concepts and introduces the notation used in the rest of the paper. Section 3 is devoted to the analysis of 2B–Consistency. Features and properties of Box–Consistency are the focus of Section 4. 3B–Consistency and Bound–Consistency are introduced in section 5. Section 6 mentions efficiency issues.

2. Interval constraint solving

This section recalls some basics of interval analysis [2, 3, 12] and formally defines a constraint system over intervals of real numbers.

2.1. Notation

We mainly use the notations suggested by Kearfott [13]. Thus, throughout, boldface will denote intervals, lower case will denote scalar quantities, and upper case will denote vectors and sets. Brackets "[.]" will delimit intervals while parentheses "(.)" will delimit vectors. Underscores will denote lower bounds of intervals and overscores will denote upper bounds of intervals. $\tilde{x}$ denotes any value in interval $\mathbf{x}$ (usually *not* the center of $\mathbf{x}$).

We will also use the following notations,which are slightly non-standard :

- $\mathcal{R}^\infty = \mathcal{R} \cup \{-\infty, +\infty\}$ denotes the set of real numbers augmented with the two infinity symbols. $\overline{\mathbb{F}}$ denotes a finite subset of $\mathcal{R}^\infty$ containing $\{-\infty, +\infty\}$. Practically speaking, $\overline{\mathbb{F}}$ corresponds to the set of floating-point numbers used in the implementation of non linear constraint solvers;

- if a is a constant in $\overline{\mathbb{F}}$, a^+ (resp. a^-) corresponds to the smallest (resp. largest) number of $\overline{\mathbb{F}}$ strictly greater (resp. lower) than a;

- f, g denote functions over the reals; $c : \mathcal{R}^n \rightarrow \mathcal{B}ool$ denotes a constraint over the reals, $\mathbf{c}$ denotes a constraint over the intervals; $Var(c)$ denotes the variables occuring in constraint c.

2.2. Interval analysis

Definition 1. [Interval]
An interval $\mathbf{x} = [\underline{x}, \overline{x}]$, with $\underline{x}$ and $\overline{x} \in \overline{\mathbb{F}}$, is the set of real numbers $\{r \in \mathcal{R} \mid \underline{x} \leq r \leq \overline{x}\}$; if $\underline{x}$ or $\overline{x}$ is the infinity symbol, then $\mathbf{x}$ is an opened interval.

$\mathcal{I}$ denotes the set of intervals and is ordered by set inclusion. $\mathcal{U}(\mathcal{I})$ denotes the set of unions of intervals.

Definition 2. [Set Extension]
Let S be a subset of $\mathcal{R}$. The *Hull* of S —denoted $\square S$— is the smallest interval I such that $S \subseteq I$.

The term "smallest subset" (w.r.t. inclusion) must be understood according to the precision of floating-point operations. In the rest of the paper, we consider —as in [14, 2]— that results of floating-point operations are outward-rounded to preserve the correctness of the computation. However, we also assume that the largest computing error when computing a bound of a variable of the initial constraint system is always smaller than one float. This hypothesis may require the use of big floats [4] when computing intermediate results.

Definition 3. [Interval Extension [20, 10]]
• $\mathbf{f} : \mathcal{I}^n \rightarrow \mathcal{I}$ is an interval extension of $f : \mathcal{R}^n \rightarrow \mathcal{R}$ iff $\forall \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{I}$: $f(\tilde{x}_1, \ldots, \tilde{x}_n) \in \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$.
• $\mathbf{c} : \mathcal{I}^n \rightarrow \mathcal{B}ool$ is an interval extension of $c : \mathcal{R}^n \rightarrow \mathcal{B}ool$ iff $\forall\ \mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathcal{I}$: $c(\tilde{x}_1, \ldots, \tilde{x}_n) \Rightarrow \mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_n})$

Similarly, $\mathbf{f}$ is the *natural* interval extension of f (see [20]) if $\mathbf{f}$ is obtained by replacing in f each constant k with the smallest interval containing k, each variable x with an interval variable $\mathbf{x}$, and each arithmetic operation with its optimal interval extension [20].
In the rest of this paper, $\mathbf{c}$ denotes the natural interval extension of c and $\oplus, \ominus, \otimes, \oslash$ denote the optimal interval extensions of $+, -, \times, /$.

We now recall a fundamental result of interval analysis with many consequences on efficiency and precision of interval constraint solving methods.

PROPOSITION 1 *[20]*
Let $\mathbf{f}$*:* $\mathcal{I}^n \rightarrow \mathcal{I}$ *be the natural interval extension of* $f : \mathcal{R}^n \rightarrow \mathcal{R}$*. If each* x_i *occurs only once in* f *then* $\square\{f(\tilde{x}_1, \ldots, \tilde{x}_n)\} = \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$ *else* $\square\{f(\tilde{x}_1, \ldots, \tilde{x}_n)\} \subseteq \mathbf{f}(\mathbf{x_1}, \ldots, \mathbf{x_n})$.

This result can be trivially extended to relations over $\mathcal{R}^n$:

PROPOSITION 2
Let $\mathbf{c} : \mathcal{I}^n \rightarrow \mathcal{B}ool$ *be the natural extension of* $c : \mathcal{R}^n \rightarrow \mathcal{B}ool$*, if each* x_i *occurs only once in* c*, then* $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_n}) \Leftrightarrow c(\tilde{x}_1, \ldots, \tilde{x}_n)$.

2.3. Interval constraint system

Definition 4. [CSP]
A CSP (Constraint System Problem) [18] is a couple (X, C) where $X = \{x_1, \ldots, x_n\}$

denotes a set of variables with associated interval domains $\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$, and $C = \{c_1, \ldots, c_m\}$ denotes a set of constraints.

$P_\emptyset$ denotes an empty CSP, i.e., a CSP with at least one empty domain. $\mathbf{X}' \subseteq \mathbf{X}$ means $\mathbf{x}'_\mathbf{i} \subseteq \mathbf{x_i}$ for all i. We define a CSP $P = (X, C)$ to be smaller than a CSP $P' = (X', C)$ if $\mathbf{X} \subseteq \mathbf{X}'$. We write $P \preceq P'$ for this relation. By convention, $P_\emptyset$ is the smallest CSP.

In the following passage, we define and discuss several kinds of consistency, and the associated filtering of a CSP P.

3. 2B–Consistency

Most of the CSP systems over intervals (e.g., [23, 3, 25, 1]) compute a relaxation of Arc–Consistency [18] called 2B–Consistency (or Hull consistency). In this section, we give the definition of 2B–consistency and explain why its computation requires a relaxation of the constraint system.

3.1. Definitions

2B–Consistency [15] states a local property on the bounds of the domains of a variable at a single constraint level. Roughly speaking, a constraint c is 2B–Consistent if, for any variable x, there exist values in the domains of all other variables which satisfy c when x is fixed to $\underline{x}$ and $\overline{x}$.

Definition 5. [2B–Consistency]
Let (X, C) be a CSP and $c \in C$ a k-ary constraint over $(x_1, \ldots, x_k)$. c is 2B–Consistent iff :
$\forall i, \mathbf{x_i} = \square\{\tilde{x}_i \mid \exists \tilde{x}_1 \in \mathbf{x_1}, \ldots, \exists \tilde{x}_{i-1} \in \mathbf{x_{i-1}}, \exists \tilde{x}_{i+1} \in \mathbf{x_{i+1}}, \ldots, \exists \tilde{x}_k \in \mathbf{x_k}$ such that $c(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i, \tilde{x}_{i+1} \ldots, \tilde{x}_k)$ holds$\}$.
A CSP is 2B–Consistent iff all its constraints are 2B–Consistent.

By definition, 2B–Consistency is weaker than Arc–Consistency. This point is illustrated in example 1.

EXAMPLE 1 *Let $P_1 = (\{x_1, x_2\}, \{x_1 = x_2 * x_2\})$ be a CSP with $\mathbf{x_1} = [1, 4], \mathbf{x_2} = [-2, 2]$. P_1 is 2B–Consistent but not arc–Consistent since there is no value in $\mathbf{x}_1$ which satisfies the constraint when $x_2 = 0$.*

Definition 6. [Closure by 2B–Consistency] [15]
The filtering by 2B–Consistency of $P = (X, C)$ is the CSP $P' = (X', C)$ such that :

- P and P' have the same solutions;
- P' is 2B–Consistent;

- $X' \subseteq X$ and the domains in X' are the largest ones for which P' is 2B–Consistent.

We note $\Phi_{2B}(P)$ the filtering by 2B–Consistency of P. In the following we will use the term *closure* by 2B–Consistency to emphasize the fact that this filtering always exists and is unique [15].

Proposition 3 states a property which is useful when comparing 2B–Consistency and Box–Consistency.

PROPOSITION 3
Let $P = (X, C)$ be a CSP such that no variable occurs more than once in any constraint of C. Let $c \in C$ be a k-ary constraint over the variables $(x_1, \ldots, x_k)$. P is 2B–Consistent iff $\forall c \in C, \forall i \in 1..k$ the following relations hold :

- $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$, *and*
- $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, (\overline{\mathbf{x_i}}^-, \overline{\mathbf{x_i}}], \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$.

Proof: Assume that both $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$ and $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, (\overline{\mathbf{x_i}}^-, \overline{\mathbf{x_i}}], \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$ hold. By proposition 2 we have :

1. $\exists \tilde{x}_1 \in \mathbf{x_1}, \ldots, \exists \tilde{x}_{i-1} \in \mathbf{x_{i-1}}, \exists x_i \in [\underline{x_i}, \underline{x_i}^+), \exists \tilde{x}_{i+1} \in \mathbf{x_{i+1}}, \ldots, \exists \tilde{x}_k \in \mathbf{x_k}$ such that $c(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i, \tilde{x}_{i+1}, \ldots, \tilde{x}_k)$ holds, and
2. $\exists \tilde{x}_1 \in \mathbf{x_1}, \ldots, \exists \tilde{x}_{i-1} \in \mathbf{x_{i-1}}, \exists x_i \in (\overline{x_i}^-, \overline{x_i}], \exists \tilde{x}_{i+1} \in \mathbf{x_{i+1}}, \ldots, \exists \tilde{x}_k \in \mathbf{x_k}$ such that $c(\tilde{x}_1, \ldots, \tilde{x}_{i-1}, \tilde{x}_i, \tilde{x}_{i+1}, \ldots, \tilde{x}_k)$ holds.

Thus, $\mathbf{x_i} = \square\{\tilde{x}_i \mid \exists \tilde{x}_1 \in \mathbf{x_1}, \ldots, \exists \tilde{x}_{i-1} \in \mathbf{x_{i-1}}, \exists \tilde{x}_{i+1} \in \mathbf{x_{i+1}}, \ldots, \exists \tilde{x}_k \in \mathbf{x_k}$ such that $c(\tilde{x}_1, \ldots, \tilde{x}_i, \ldots, \tilde{x}_k)$ holds $\}$.
The counterpart results from the definition of 2B–Consistency. ■

3.2. Computing 2B–Consistency

2B–Consistency is enforced by narrowing the domains of the variables. Using the above notations, the scheme of the standard interval narrowing algorithm —derived from AC3 [18]– can be written down as in figure 1. `IN` implements the computation of the closure by 2B-consistency of a CSP $P = (X, C)$. $narrow(c, \mathbf{X})$ is a function which prunes the domains of variables $Var(c)$ until c is 2B–consistent.

The approximation of the projection functions is the basic tool for the narrowing of domains in $narrow(c, \mathbf{X})$. Let c be a k-ary constraint over $X = (x_1, \ldots, x_k)$: for each i in $1..k$, $\pi_i(c, \mathbf{X})$ denotes the projection over x_i of the solutions of c in the space delimited by $\mathbf{X}$.

Definition 7. [projection of a constraint]
$\pi_i(c, \mathbf{X}) : (C, I^k) \rightarrow \mathcal{U}(\mathcal{I})$ is the projection of c on x_i iff $\pi_i(c, \mathbf{X}) = \{\tilde{x}_i \mid \exists (\tilde{x}_1, \ldots,$

```
1.   IN(in C, inout X)
2.   Queue ← C;
3.   while Queue ≠ ∅
4.       c ← POP Queue;
5.       X' ← narrow(c, X);
6.       if X' ≠ X then
7.           X ← X';
8.           Queue ←Queue ∪ {c' ∈ C | Var(c) ∩ Var(c') ≠ ∅}
9.       endif
10.  endwhile
```

Figure 1. Algorithm IN

$\tilde{x}_{i-1}, \tilde{x}_{i+1}, \ldots, \tilde{x}_k) \in \mathbf{x_1} \times \ldots \times \mathbf{x_{i-1}} \times \mathbf{x_{i+1}}, \ldots \times \mathbf{x_k}$ such that $c(\tilde{x}_1, \ldots, \tilde{x}_i, \ldots, \tilde{x}_k)$ holds}.

Definition 8. [approximation of the projection]
$AP_i(c, \mathbf{X}) : (C, \mathcal{I}^k) \rightarrow \mathcal{I}$ is an approximation of $\pi_i(c, \mathbf{X})$ iff $AP_i(c, \mathbf{X}) = \Box\, \pi_i(c, \mathbf{X}) = [\min\ \pi_i(c, \mathbf{X}), \max\ \pi_i(c, \mathbf{X})]$. In other words, $AP_i(c, \mathbf{X})$ is the smallest interval encompassing projection $\pi_i(c, \mathbf{X})$.

The following proposition trivially holds :

PROPOSITION 4
Constraint c is 2B–Consistent on $\mathbf{X}$ *iff for all i in* $\{1, \ldots, k\}$, $\mathbf{x_i} = AP_i(c, \mathbf{X})$.

In general, AP_i cannot be computed efficiently because it is difficult to define functions `min` and `max`, especially when c is not monotonic. For instance, if variable x has multiple occurrences in c, defining these functions would require x to be isolated [1]. Since such a symbolic transformation is not always possible, this problem is usually solved by decomposing the constraint system into a set of primitive constraints for which the AP_i can easily be computed [17]. Primitive constraints are generated syntactically by introducing new variables.

Definition 9. [decomposition of a constraint system]
Let $P = (X, C)$ be a CSP and $c \in C$ a constraint. We define $\mathcal{M}_c \subseteq X$ as the set of variables having multiple occurrences in c. $decomp(c)$ is the set of constraints obtained by substituting in c each occurrence of variables $x \in \mathcal{M}_c$ with a new variable y with domain $\mathbf{y} = \mathbf{x}$ and by adding a constraint $x = y$. $New_{(x,c)}$ is the set of new variables introduced to remove multiple occurrences of variable x in c, $X_{New} = \bigcup\{New_{(x,c)} \mid x \in X \text{ and } c \in C\}$. P_{decomp} is the CSP (X', C') where $X' = X \cup X_{New}$, and $C' = \{decomp(c) \mid c \in C\}$.

Decomposition does not change the semantics of the constraint system : P and P_{decomp} have the same solutions since P_{decomp} just results from a rewriting[2] of P. However, a *local* consistency like Arc–Consistency is not preserved by such a rewriting. Indeed, decomposition reduces the scope of local consistency filtering algorithms. Thus, P_{decomp} is a *relaxation* of P when computing a relaxation of Arc–Consistency.

EXAMPLE 2 (DECOMPOSITION OF THE CONSTRAINT SYSTEM) *Let* $c : x_1 + x_2 - x_1 = 0$ *be a constraint and* $\mathbf{x_1} = [-1, 1]$, $\mathbf{x_2} = [0, 1]$ *the domains of* x_1 *and* x_2. *Since* x_1 *appears twice in* c, *its second occurrence will be replaced with a new variable* x_3 : $decomp(c) = \{x_1 + x_2 - x_3 = 0, x_1 = x_3\}$.

In this new constraint system, each projection can easily be computed with interval arithmetic. For instance, $AP_1(x_1 + x_2 - x_3 = 0, (\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}))$ *is* $\mathbf{x_1} \cap (\mathbf{x_3} \ominus \mathbf{x_2})$. *However, this decomposition increases the locality problem: the first constraint is checked independently of the second one and so* x_1 *and* x_3 *can take distinct values. More specifically, the initial constraint* c *is not 2B–Consistent since there is no value of* x_1 *which satisfies* c *when* $x_2 = 1$. *On the contrary,* $decomp(c)$ *is 2B–Consistent since the values* $x_1 = -1$ *and* $x_3 = 0$ *satisfy* $x_1 + x_2 - x_3 = 0$ *when* $x_2 = 1$. *On the initial constraint, 2B–Consistency reduces* $\mathbf{x_2}$ *to* $[0, 0]$ *while it yields* $\mathbf{x_1} = [-1, 1], \mathbf{x_2} = [0, 1]$ *for* $decomp(c)$.

Remark. Like almost all other examples in this paper, Example 2 can be trivially simplified. However, the reader can more easily check partial consistencies on such examples than on non–linear constraints where the same problems occur.

4. Box–Consistency

Box–Consistency [2, 11] is a coarser relaxation of Arc–Consistency than 2B–Consistency. It mainly consists of replacing every existentially quantified variable but one with its interval in the definition of 2B–Consistency. Thus, Box–Consistency generates a system of univariate interval functions which can be tackled by numerical methods such as Newton. Contrary to 2B–Consistency, Box–Consistency does not require any constraint decomposition and thus does not amplify the locality problem. Moreover, Box–Consistency can tackle some dependency problems when each constraint of a CSP contains only one variable which has multiple occurrences.

4.1. Definition and properties of Box–Consistency

Definition 10. [Box–Consistency]
Let (X, C) be a CSP and $c \in C$ a k-ary constraint over the variables $(x_1, \ldots, x_k)$. c is Box–Consistent if, for all x_i the following relations hold :
1. $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x_i}}, \underline{\mathbf{x_i}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$,
2. $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, (\overline{\mathbf{x_i}}^-, \overline{\mathbf{x_i}}], \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$.

Closure by Box–Consistency of P is defined similarly to closure by 2B–Consistency of P, and is denoted by $\Phi_{Box}(P)$.

PROPOSITION 5
$\Phi_{2B}(P) \preceq \Phi_{Box}(P)$ and $\Phi_{2B}(P) \equiv \Phi_{Box}(P)$ when no variable occurs more than once in the constraints of C.

Proof: From the definitions of 2B–Consistency, Box–Consistency and interval extension of a relation, it results that $\Phi_{2B}(P) \preceq \Phi_{Box}(P)$. By proposition 2 the equivalence holds when no variable occurs more than once in the constraints of C. ■

It follows that any CSP which is 2B–Consistent is also Box–Consistent. On the contrary a CSP which is Box–Consistent may not be 2B–Consistent (see example 3).

EXAMPLE 3 *Example 2 is not 2B–Consistent for x_2 but it is Box–Consistent for x_2 since $([-1,1] \oplus [0,0^+] \ominus [-1,1]) \cap [0,0]$ and $([-1,1] \oplus [1^-,1] \ominus [-1,1]) \cap [0,0]$ are non-empty.*

Of course, the decomposition of a constraint system amplifies the limit due to the local scope of 2B–Consistency. As a consequence, 2B–Consistency on the decomposed system yields a weaker filtering than Box–Consistency on the initial system :

PROPOSITION 6
$\Phi_{Box}(P) \preceq \Phi_{2B}(P_{decomp})$

Proof: The different occurrences of the same variable are connected by the existential quantifier as stated in the definition of the 2B–Consistency. However, the decomposition step breaks down the links among these different occurrences and generates a CSP P_{decomp} which is a relaxation of P for the computation of a local consistency. It follows that $\Phi_{Box}(\mathrm{P}) \preceq \Phi_{Box}(P_{decomp})$. By proposition 5 we have : $\Phi_{Box}(P_{decomp}) \equiv \Phi_{2B}(P_{decomp})$, and thus $\Phi_{Box}(P) \preceq \Phi_{2B}(P_{decomp})$ ■

EXAMPLE 4 *Let c be the constraint $x_1 + x_2 - x_1 - x_1 = 0$ where $\mathbf{x_1} = [-1,1]$ and $\mathbf{x_2} = [0.5,1]$. c is not Box–Consistent since $[-1,-1^+] \oplus [0.5,1] \ominus [-1,-1^+] \ominus [-1,-1^+] \cap [0,0]$ is empty. But decomp(c) is 2B–Consistent for $\mathbf{x_1}$ and $\mathbf{x_2}$.*

Box–Consistency can tackle some dependency problems in a constraint c which contains only one variable occuring more than once. More precisely, Box–Consistency enable us to reduce the domain $\mathbf{x}$ if variable x occurs more than once in c and if $\mathbf{x}$ contains inconsistent values. For instance, in example 4, filtering by Box–consistency reduces $\mathbf{x_1}$ because value -1 of $\mathbf{x_1}$ has no support in $\mathbf{x_2}$.
However, Box–Consistency may fail to handle the dependency problem when the inconsistent values of constraint c are in the domain of variable x_i while a variable x_j $(j \neq i)$ occurs more than once in c. For instance, in example 2, value 1 of $\mathbf{x_2}$ has no support in $\mathbf{x_1}$ but Box–Consistency fails to detect the inconsistency because $[-1,1] \oplus [1^-,1] \ominus [-1,1] \cap [0,0]$ is not empty.

4.2. Computing Box–Consistency

The Box–Consistency filtering algorithm proposed in [2, 28, 29] is based on an iterative narrowing operation using the interval extension of the Newton method. Computing Box–Consistency follows the generic algorithm `IN` (see figure 1) used for computing 2B–Consistency. The function $narrow(c, \mathbf{X})$ prunes the domains of the variables of c until c is Box–consistent. Roughly speaking, for each variable x of constraint c, an interval univariate function $\mathbf{f_x}$ is generated from c by replacing all variables but x with their intervals. The narrowing process consists of finding the leftmost and rightmost zeros of $\mathbf{f_x}$. Figure 2 shows function `LNAR` which computes the leftmost zero of $\mathbf{f_x}$ for initial domain I_x of variable x (this procedure is given in [29]).

```
1.  function LNAR (IN: f_x, x, RETURN: Interval)
2.      r ← x̄
3.      if    0 ∉ f_x(x) then return ∅
4.          else    i ← NEWTON(f_x, x)
5.              if    0 ∈ f_x([i, i⁺]) then return [i, r]
6.                  else    SPLIT(i, i_1, i_2)
7.                      l_1 ← LNAR(f_x, i_1)
8.                      if    l_1 ≠ ∅ then return [l_1, r]
9.                          else return [LNAR(f_x, i_2), r]
10.                     endif
11.             endif
12.     endif
```

Figure 2. Function `LNAR`

Function `LNAR` first prunes interval $\mathbf{x}$ with function `NEWTON` which is an interval version of the classical Newton method. However, depending on the value of $\mathbf{x}$, Newton may not reduce $\mathbf{x}$ enough to make $\mathbf{x}$ Box–Consistent. So, a split step is applied in order to ensure that the left bound of $\mathbf{x}$ is actually a zero. Function `SPLIT` divides interval $\mathbf{i}$ in two intervals $\mathbf{i_1}$ and $\mathbf{i_2}$, $\mathbf{i_1}$ being the left part of the interval. The splitting process avoids the problem of finding a safe starting box for Newton (see [11]). As mentioned in [29], even if $\mathbf{f_x}$ is not differentiable, the function `LNAR` may find the leftmost zero thanks to the splitting process (in this case, the call to function `NEWTON` is just ignored). Notice that Box–consistency can be computed in such a way because it is defined on interval constraints whereas the existential quantifiers in the definition of 2B–consistency require the use of projection functions.

5. 3B–Consistency and Bound–Consistency

2B–Consistency and Box–Consistency are only partial consistencies which are often too weak for computing an relevant superset of solutions of a CSP. In the same way that Arc–Consistency has been generalized to higher consistencies (e.g., path consistency [18]), 2B–Consistency and Box–Consistency can be generalized to higher order consistencies [15].

5.1. 3B–Consistency

Definition 11. [3B–Consistency] [15]
Let $P = (X, C)$ be a CSP and x a variable of X. Let also :

- $P_{\mathbf{x} \leftarrow [\underline{x}, \underline{x}^+)}$ be the CSP derived from P by substituting $\mathbf{x}$ with $[\underline{x}, \underline{x}^+)$;
- $P_{\mathbf{x} \leftarrow (\overline{x}^-, \overline{x}]}$ be the CSP derived from P by substituting $\mathbf{x}$ with $(\overline{x}^-, \overline{x}]$.

$\mathbf{x}$ is 3B–Consistent iff $\Phi_{2B}(P_{\mathbf{x} \leftarrow [\underline{x}, \underline{x}^+)}) \neq P_\emptyset$ and $\Phi_{2B}(P_{\mathbf{x} \leftarrow (\overline{x}^-, \overline{x}]}) \neq P_\emptyset$. A CSP is 3B–Consistent iff all its domains are 3B–Consistent.

It results from definition 11 that any CSP which is 3B–Consistent is also 2B–Consistent [15]. The generalization of the 3B–Consistency to kB–Consistency is straightforward and is given in [15, 17].

3B–Consistency is less local than 2B–Consistency or Box–Consistency. Proposition 7 shows that 3B–Consistency always prunes more strongly than Box–Consistency, even if 3B–Consistency is achieved on the decomposed system and Box–consistency on the initial system.

PROPOSITION 7
Let $P = (X, C)$ be a CSP. If P_{decomp} is 3B–Consistent then P is Box–Consistent.

Proof: Since Box-consistency is a local consistency we just need to show that the property holds for a single constraint.
Assume c is a constraint over $(x_1, ..., x_k)$, x is one of the variables occuring more than once in c and $New_{(x,c)} = (x_{k+1}, \ldots, x_{k+m})$ is the set of variables introduced for replacing the multiple occurrences of x in c. Suppose that P_{decomp} is 3B–Consistent for $\mathbf{x}$.
Consider P_1, the CSP derived from P_{decomp} by reducing domain $\mathbf{x}$ to $[\underline{x}, \underline{x}^+)$. P_1 is 2B–Consistent for $\mathbf{x}$ and thus the domain of all variables in $New_{(x,c)}$ is reduced to $[\underline{x}, \underline{x}^+)$; this is due to the equality constraints added when introducing new variables. From proposition 3, it results that the following relation holds:
$\mathbf{c'}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k}, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+), \ldots, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+), \mathbf{x_{k+m+1}}, \ldots, \mathbf{x_n})$
$\mathbf{c'}$ is the very same syntactical expression as $\mathbf{c}$ (where some variables have been renamed).
$(\mathbf{x_{k+m}}, \ldots, \mathbf{x_n})$ are the domains of the variables introduced for replacing the multiple occurrences of $\mathcal{M}_c \setminus \{x\}$. As the natural interval extension of a constraint is

defined over the intervals corresponding to the domains of the variables, relation $\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})$ holds too.

The same reasoning can be applied when x is replaced with its upper bound $(\overline{x}^-, \overline{x}]$. So we conclude that $\mathbf{x}$ is also Box–Consistent. ■

EXAMPLE 5 *Let $C = \{x_1 + x_2 = 100, x_1 - x_2 = 0\}$ and $\mathbf{x_1} = [0, 100], \mathbf{x_2} = [0, 100]$ be the constraints and domains of a given CSP P.*
$\Phi_{3B}(P_{decomp})$ reduces the domains of x_1 and x_2 to the interval [50,50] whereas $\Phi_{Box}(P)$ does not achieve any pruning (P is Box–Consistent).

The following proposition is a direct consequence of proposition 7 :

PROPOSITION 8
$\Phi_{3B}(P_{decomp}) \preceq \Phi_{Box}(P)$.

Thus, 3B–Consistency allows us to tackle at least the same dependency problems as Box–consistency. However, 3B–Consistency is not effective enough to tackle the dependency problem in general (see example 6).

EXAMPLE 6 *Let c be the constraint $x_1 * x_2 - x_1 + x_3 - x_1 + x_1 = 0$ where $\mathbf{x_1} = [-4, 3], \mathbf{x_2} = [1, 2]$ and $\mathbf{x_3} = [-1, 5]$. $decomp(c) = \{x_1 * x_2 - x_4 + x_3 - x_5 + x_6 = 0, x_1 = x_4 = x_5 = x_6\}$. c is not 2B–Consistent since there are no values in $\mathbf{x_1}$ and $\mathbf{x_2}$ which verify the relation when $x_3 = 5$.*
However, $decomp(c)$ is 3B–Consistent. Indeed, the loss of the link between the two occurrences of x_1 prevents the pruning of x_3.

A question which naturally arises is that of the relation which holds between Φ_{2B}(P) and $\Phi_{3B}(P_{decomp})$: example 7 shows that Φ_{2B}(P) $\preceq \Phi_{3B}(P_{decomp})$ does not hold and example 6 shows that $\Phi_{3B}(P_{decomp}) \preceq \Phi_{2B}(P)$ does not hold, even if only one variable occurs more than once in each constraint of P. It follows that no order relation between $\Phi_{3B}(P_{decomp})$ and Φ_{2B}(P) can be exhibited.

EXAMPLE 7 *Let P be a CSP defined by $C = \{x_1 + x_2 = 10; x_1 + x_1 - 2x_2 = 0\}$ where $\mathbf{x_1} = \mathbf{x_2} = [0, 10]$. $decomp(x_1 + x_1 - 2x_2 = 0) = \{x_1 + x_3 - 2x_2 = 0, x_3 = x_1\}$.*
P is 2B–Consistent but P_{decomp} is not 3B–Consistent : Indeed, when x_1 is fixed to 10, $\Phi_{2B}(P_{\mathbf{x_1} \leftarrow [10^-, 10]}) = P_\emptyset$ since $\mathbf{x_2}$ is reduced to $\emptyset$. In this case, the link between x_1 and x_3 is preserved and 3B–Consistency reduces $\mathbf{x_2}$ to [5,5].

5.2. Bound–Consistency

Bound-consistency was suggested in [17] and was formally defined in [28]. Informally speaking, Bound-consistency applies the principle of 3B–Consistency to Box–Consistency : it checks whether Box–Consistency can be enforced when the domain of a variable is reduced to the value of one of its bounds in the whole system.

Definition 12. [Bound–Consistency]
Let (X, C) be a CSP and $c \in C$ a k-ary constraint over the variables $(x_1, \ldots, x_k)$. c is Bound–Consistent if for all x_i, the following relations hold :
1. $\Phi_{Box}(\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, [\underline{\mathbf{x}}, \underline{\mathbf{x}}^+), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})) \neq P_\emptyset$,
2. $\Phi_{Box}(\mathbf{c}(\mathbf{x_1}, \ldots, \mathbf{x_{i-1}}, (\overline{\mathbf{x}}^-, \overline{\mathbf{x}}), \mathbf{x_{i+1}}, \ldots, \mathbf{x_k})) \neq P_\emptyset$.

Since $\Phi_{Box}(\mathrm{P}) \preceq \Phi_{2B}(P_{decomp})$ it is trivial to show that $\Phi_{Bound}(P) \preceq \Phi_{3B}(P_{decomp})$. Bound–Consistency achieves the same pruning as 3B–Consistency when applied to examples 5 and 2.

6. Efficiency issues

The aim of numerical CSP is not to compute partial consistencies but to find accurate solutions; that is, either small intervals containing isolated solutions, or intervals which tightly encompass sets of continuous solutions. Thus, in practical systems (e.g., `Numerica` [28], `PROLOG IV` [25]), partial consistencies are combined with several search heuristics and splitting techniques. Experimental results of `Numerica` and `Newton` are very impressive. However it is difficult to draw a conclusion from the published benchmarks because these systems differ in several critical points :

- They use different splitting heuristics;
- There are significant variations in the implementation of the filtering algorithms (precision parameters in order to force an early halt to the propagation process, constraint ordering, detection of cycles ..);
- They use different implementation languages (`Prolog`, `C`, ...).

So we limit the discussion to a brief examination of three key points which help to better understand the performances of the different systems:

1. **Cost of the basic narrowing operator** :
   Performing interval newton method on univariate functions is more expensive than computing projection functions on primitive constraints. For instance, let $C = \{x^2 = 2\}$ and $\mathbf{x} = [1, 10]$. Box–consistency requires 6 narrowing steps with the Newton method (about 100 interval operations) whereas 2B–consistency only requires the computation of one relational square root operation and one intersection over intervals [9]. Thus, 2B is more efficient than Box on problems where the projection functions compute an accurate result.

   The gain of performance due to accurate projection functions is well illustrated using the pentagon problem. This problem consists of finding the coordinates of five points of a circle such that these points define a convex pentagon. The constraint system consists of five quadratic equations. To avoid an infinite number of solutions, the first point is given, and the five points are ordered to avoid symmetrical solutions. On this example, Bound–Consistency and 3B–Consistency achieve the same pruning. According to [1] Box is about forty times slower than 2B on this example.

2. **Expansion of the constraint system** :
   Decomposition of the initial constraint system may generate a huge number of primitive constraints when variables occur more than once. For instance, consider the classical "Broyden 160" example (160 initial variables, 160 constraints). Box–consistency will generate 160 variables, 1184 univariate functions whereas 2B–consistency will generate 2368 variables, 6944 ternary projection functions.
   From a practical point of view, 2B–Consistency is seriously weakened by the decomposition required for computing the narrowing functions. On the other hand, the univariate functions generated by Box–Consistency can be handled very efficiently using Newton-like methods.

3. **Precision of the computation** :
   For a fixed final precision, the efficiency of the computation may strongly depend on the accuracy of the partial consistency filtering algorithm. For instance, consider again the resolution of the "Broyden 160" problem by combining Box–consistency filtering and a domain splitting strategy. If the final intervals have to of a size smaller than or equal to 10^{-8}, the computation is about 10 times faster with a coarse relaxation of Box–consistency than with an accurate one [9].

It appears that the following approaches are promising:

- Combining different partial consistencies [9];
- Pre-processing of the constraints (e.g., symbolic transformations)
- Intelligent search strategies (e.g., use of extrapolation techniques before starting a costly filtering process [16], dynamic choice of the filtering precision).

7. Conclusion

This paper has investigated the relations among 2B–Consistency, 3B–Consistency, Box–Consistency and Bound–Consistency. The main result is a proof of the following properties :

$$\mathbf{\Phi_{Bound}(P)} \preceq \mathbf{\Phi_{3B}(P_{decomp})} \preceq \mathbf{\Phi_{Box}(P)}$$
$$\mathbf{\Phi_{2B}(P)} \preceq \mathbf{\Phi_{Box}(P)} \preceq \mathbf{\Phi_{2B}(P_{decomp})}$$
$$\mathbf{\Phi_{3B}(P_{decomp})} \text{ and } \mathbf{\Phi_{2B}(P)} \text{ are not comparable}$$

The advantage of Box–Consistency is that it generates univariate functions which can be tackled by numerical methods such as Newton, and which do not require any constraint decomposition. On the other hand, 2B–Consistency algorithms require a decomposition of the constraints with multiple occurrences of the same variable. This decomposition increases the limitations due to the local nature of 2B–Consistency. As expected, higher consistencies — e.g., 3B–Consistency and

Bound–Consistency — can reduce the drawbacks due to the local scope of the inconsistency detection.

Efficiency of the filtering algorithms is a critical issue, but it is difficult to draw a conclusion from the published benchmarks. Further experimentation combining these different partial consistencies and various search techniques is required to better understand their advantages and drawbacks and to define the class of application in which each of them is most relevant.

Acknowledgments

Thanks to Olivier Lhomme, Christian Bliek and Bertrand Neveu for their careful reading and helpful comments on earlier drafts of this paper. Thanks also to Frédéric Goualard, Laurent Granvilliers, and Arnold Neumaier for interesting discussions.
Thanks also go to a referee for his many comments, which hopefully led to significant improvements of this paper.

Notes

1. B. Faltings [7] has recently introduced a new method for computing the projection without defining projection function. However, this method requires a complex analysis of constraints in order to find extrema.
2. In practice, c is decomposed into binary and ternary constraints for which projection functions are straightforward to compute. Since there are no multiple occurrences in $decomp(c)$ and interval calculus is associative, this binary and ternary constraint system has the same solutions as P_{decomp}.

References

1. F. Benhamou, F. Goualard, and L. Granvilliers. Programming with the DecLIC Language *In proceedings of the second workshop on Interval Constraints.* October 1997, Port-Jefferson, NY, USA.
2. F. Benhamou, D. Mc Allester, and P. Van Hentenryck. CLP(Intervals) Revisited. in *Proc. Logic Programming : Proceedings of the 1994 International Symposium*, MIT Press, (1994).
3. F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*,Vol. 32, pp. 1-24, 1997.
4. R.P. Brent. A FORTRAN multiple-precision arithmetic package *ACM Trans. on Math. Software*, 4, no 1, 57-70, 1978.
5. J.C. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2) :125–149, 1987.
6. H.Collavizza, F.Delobel, M. Rueher A Note on Partial Consistencies over Continuous Domains Solving Techniques *Proc. CP98 (Fourth International Conference on Principles and Practice of Constraint Programming)*, LNCS 1520, Springer Verlag 1998.
7. B. Faltings. Arc–consistency for continuous variables. *Artificial Intelligence*, vol. 65, pp. 363–376, 1994.
8. E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21 :958–966, November 1978.
9. L. Granvilliers. On the combination of Box–consistency and Hull-consistency. Workshop ECAI on non binary-constraints, Brighton, United Kingdom,1998
10. E. Hansen. Global optimization using interval analysis. *Marcel Dekker*, NY, 1992.

11. H. Hong, V. Stahl . *Safe Starting Regions by Fixed Points and Tightening. Computing*, 53 :323–335, 1994.
12. E. Hyvőnen. Constraint reasoning based on interval arithmetic : the tolerance propagation approach. *Artificial Intelligence*, vol. 58, pp. 71–112, 1992.
13. R. Baker Kearfott. Rigorous Global Search: Continuous Problems. *Kluwer Academic Publishers*, Dordrecht, Netherlands, 1996
14. J. H. M. Lee and M. H. van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16 :3–4, pp.255–276, 1993.
15. O. Lhomme. Consistency techniques for numeric CSPs. in *Proc. IJCAI93, Chambery, (France)*, pp. 232–238, (August 1993).
16. Y. Lebbah, O. Lhomme. Acceleration methods for numeric CSPs AAAI, MIT Press, 1998.
17. O. Lhomme and M. Rueher. Application des techniques CSP au raisonnement sur les intervalles. *RIA (Dunod)*,vol. 11 :3, pp. 283–312, 1997.
18. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.
19. U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2) : 95–132, 1974.
20. R. Moore, Interval Analysis. *Prentice Hall*, 1966.
21. A. Neumaier. Interval methods for systems of equations. *Cambridge University Press*, 1990.
22. A. Neumaier, S. Dallwig and H. Schichl, GLOPT - A Program for Constrained Global Optimization. in I. Bomze et al., eds., *Developments in Global Optimization*, Kluwer, pp. 19-36, Dordrecht 1997.
23. W.J. Older and A. Vellino. Extending prolog with constraint arithmetic on real intervals. In *Proc. of IEEE Canadian conference on Electrical and Computer Engineering*. IEEE Computer Society Press, 1990.
24. J-F. Puget and P. Van Hentenryck. A Constraint Satisfaction Approach to a Circuit Design Problem *Journal of Global Optimization*, 13: 75-93, 1998
25. Prologia *PrologIV Constraints inside*. Parc technologique de Luminy - Case 919 13288 Marseille cedex 09 (France), 1996.
26. M. Rueher, C. Solnon. Concurrent Cooperating Solvers within the Reals. *Reliable Computing*. Kluwer Academic Publishers, Vol.3 :3, pp. 325-333, 1997.
27. E. Tsang. Foundations of Constraint Satisfaction. *Academic Press*, 1993.
28. P. Van Hentenryck, Y. Deville, and L. Michel. *Numerica. A modeling language for global optimization*. MIT Press, 1997.
29. P. Van Hentenryck, D. McAllester, and D. Kapur. *Solving Polynomial Systems Using a Branch and Prune Aprroach. SIAM Journal on Numerical Analysis*, 34(2), April 1997.

V.5.2 Extending consistent domains of numeric CSP

H.Collavizza, F.Delobel, M. Rueher. Extending consistent domains of numeric CSP. IJCAI-99, Stockholm, Sweden, 31 July - 6 August 1999 .

Extending consistent domains of numeric CSP

Hélène Collavizza, François Delobel, Michel Rueher
Université de Nice–Sophia-Antipolis, I3S — ESSI
930, route des Colles - B.P. 145, 06903 Sophia-Antipolis, France
{helen,delobel,rueher}@essi.fr

Abstract

This paper introduces a new framework for extending consistent domains of numeric CSP. The aim is to offer the greatest possible freedom of choice for one variable to the designer of a CAD application. Thus, we provide here an efficient and incremental algorithm which computes the maximal extension of the domain of one variable. The key point of this framework is the definition, for each inequality, of an *univariate* extrema function which computes the left most and right most solutions of a selected variable (in a space delimited by the domains of the other variables). We show how these univariate extrema functions can be implemented *efficiently*. The capabilities of this approach are illustrated on a ballistic example.

1 Introduction

This paper introduces a new framework for extending the domain of one variable in a consistent CSP [1] which is defined by a set of *non-linear constraints over the reals*. The aim is to offer the greatest freedom of choice of possible values for a variable to the *designer* of a CAD application. For example, one starts from the knowledge of a solution and tries to widen the variations of a variable. This problem occurs in a large class of electro-mechanical engineering and civil engineering applications, where extending the domain of a variable permits the tolerance of any associated component to be enlarged, and therefore to lower the cost of this component. These problems are often under-constrained. So, what the user wants to know is a subset of the solutions. For these applications, classical methods (e.g.,[7; 10]), based on local consistencies and domain splitting, cannot ensure that a solution exists inside the arbitrarily small intervals they compute. Moreover, domain splitting is ineffective if the solution set is not a finite set of isolated solutions but a collection of intervals.

The framework we introduce here allows one to enlarge the domain of a variable while preserving the consistency of the CSP. Sam-Haroud and Faltings [9] have proposed an approach for computing safe solutions of non-linear constraint systems. Roughly speaking, they fill up the solution space with a set of consistent boxes[2]. Their approach could be used to extend the domain of one variable. However, the underlying costs in computation time and space are exponential.
The framework we introduce here is less general but it can be implemented efficiently. Before going into the details, let us outline our framework in very general terms. The main steps of the right extension[3] of the domain of a variable are:

1. Searching for a subset of the solution space; this solution space may be reduced to a single point;
2. Selecting of the variable the domain of which has to be extended;
3. Defining for each inequality of an extrema function that computes the left most solution of the selected variable in a space delimited by the domains of the other variables;
4. Finding the smallest solution of all extrema functions.

The following example illustrates this process.

[1] An introduction to CSP and numeric CSP can be found in [4; 7].

[2] Their approach is based upon a classical method used in graphical computing for image synthesis (composition of shapes, of scenes) known as the 2^k trees. The key idea is to classify portions of space in three categories: the black shapes contain no solution at all, the gray shapes contain solutions, but also contain points which are not solutions, and finally, the white shapes contain only points which are solution. The gray shapes are split into smaller one that are again classified into black, white and gray shapes; the decomposition process stops when the size of the shapes becomes smaller than a given value.

[3] Throughout this paper, we will only consider the right extension since the left one can be computed in a symmetrical way.

Example 1 *Let us consider the behavior of an electrical shunt motor, the speed of which may be changed. The maximum speed can be up to 3 times the value of the minimum speed. We only consider two parameters of the motor: the torque C_u, and the rotation speed N. The motor cannot use more than a given power : $N * C_u \leq P_{max}$. Moreover, the motor cannot operate above a given speed and torque: $N \in [1,3]$, $C_u \in [0,4]$.*
We know that the motor is working efficiently for every tuple of values $D_{C_u} \times D_N$ in $[0,1] \times [1,2]$ when $P_{max} = 5$. What we want to compute is the maximum range of values of the torque which is safe with this motor. In other words, we are looking for the maximum domain D_{C_u} such that every tuple in $D_{C_u} \times D_N$ is a solution of the constraint system.
Now, consider equation $N * C_u = 5$ in the space delimited by $N \in [1,3]$, $C_u \in [0,4]$. Its left most solution is the point defined by $C_u = 2.5$ and $N = 2$; this point is obviously an upper bound of the domain D_{C_u}.

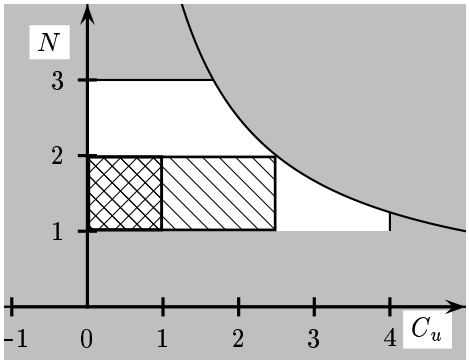


Figure 1: Relation between N and C_u

An initial subset of the solution space can often be found by experimentation. Note that the solution space may be reduced to a single point and the domains of the different variables may successively be extended.
The definition of the univariate extrema functions is a key point of our approach. Optimal univariate extrema functions can trivially be computed for the so-called primitive constraints. For non–primitive constraints, the methods used for computing Box–consistency [1] provide an efficient way to compute a safe approximation of univariate extrema functions.
To define formally the extension of the domain of a variable, we introduce an "internal" consistency, named *i–consistency*, which ensures that every tuple in the Cartesian product of the variable domains is a solution of the constraint system. i–consistency should not be mistaken with arc consistency or approximations of arc consistency [3] (e.g. 2B–consistency[7], Box–consistency[1]). Those consistencies define regions containing all the solutions (and possibly tuples which are not solution) whereas i–consistency defines a region which is a subset of the set of solutions. Figure 2 shows the relations between these different families of consistencies. Roughly speaking, the smallest external box is the best approximation which can be computed by approximations of arc consistency over continuous domains.

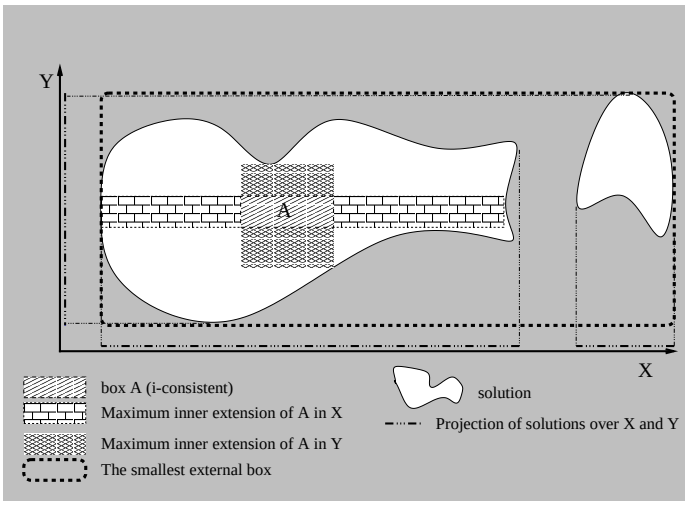


Figure 2: Relations between i–consistency and some partial consistencies

Outline of the paper: Section 2 introduces the notation and recalls the basics on CSP over continuous domains which are needed in the rest of the paper. Section 3 is devoted to the description of the i–consistent extension process. Extrema functions are formally defined and an efficient algorithm is introduced. Section 4 outlines the capabilities of our approach on a ballistic example.

2 Preliminaries

2.1 Notation

We use the following notations, possibly subscripted:

- x, y, z denote variables over the reals;
- u, v denote real constants;
- f, g denote functions over the reals;
- c denotes a constraint over the reals;

The next subsection recalls a few notions of numeric CSP; Details can be found in [2; 10; 3].

2.2 Interval constraint system

A k-ary constraint c is a relation over the reals.

Definition 1 (Interval) *Let $\overline{I\!F}$ denote a finite subset of $\mathcal{R}$ augmented with the two infinity symbols $\{-\infty, +\infty\}$. An interval $[a,b]$ with $a, b \in \overline{I\!F}$ is the set of real numbers $\{r \in \mathcal{R} \mid a \leq r \leq b\}$.*

Definition 2 (CSP)
A CSP [8] is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ denotes a set of variables, $\mathcal{D} = \{D_{x_1}, \ldots, D_{x_n}\}$ denotes a set of domains, D_{x_i} being the interval containing all

acceptable values for x_i, and $\mathcal{C} = \{c_1, \ldots, c_m\}$ denotes a set of constraints[4].

$\vec{\mathcal{D}}$ denotes the Cartesian product $D_{x_1} \times \ldots, \times D_{x_n}$. $\vec{v}$ denotes the tuple $(v_{x_1}, \ldots, v_{x_n})$ such that $\vec{v} \in \vec{\mathcal{D}}$. $\pi_x(\vec{v})$ denotes the projection over x of $\vec{v}$. $\underline{D_i}$ (resp. $\overline{D_i}$) denotes the lower bound (resp. upper bound) of the interval D_i.

Definition 3 (k-box) *A k-box $I_1 \times \ldots \times I_k$ is the part of a k-dimension space defined by the Cartesian product of intervals $(I_1, \ldots, I_k)$*

By construction, all the k-boxes are convex.

2.3 Local consistencies

Local consistencies over continuous domains are based on arc consistency[8] which was originally defined for finite domains. This section introduces two local consistencies that will be used in the rest of the paper.

Definition 4 (arc consistency) *A CSP P $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is arc–consistent iff:* $\forall D_x \in \mathcal{D}$, $\forall v_x \in D_x$, $\forall c \in \mathcal{C}$: $\exists\, \vec{v} \in \vec{\mathcal{D}} \mid \pi_x(\vec{v}) = v_x \wedge c(\vec{v})$

Davis ([4]) has studied the application of the Waltz algorithm ([12]) over continuous domains and has shown important theoretical limitations. The Waltz algorithm was then extended by Faltings ([5; 6]) in order to deal with ternary constraints defined by continuous and differentiable curves.

Definition 5 (Set Extension) *Let S be a subset of $\mathcal{R}$. The approximation of S —denoted* `hull`*— is the smallest interval I such that $S \subseteq I$.*

2.4 Box–consistency

Roughly speaking, Box–consistency [1; 10] is a local consistency over continuous domains which computes a safe approximation of the solution of each variable involved in a given constraint.

Definition 6 (Box–consistency) *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP and $c \in \mathcal{C}$ a k-ary constraint over the variables $(x_1, \ldots, x_k)$. c is Box–consistent if, for all x_i in $\{x_1, \ldots, x_k\}$ such that $D_{x_i} = [a, b]$, the following relations hold :*
1. $C(D_{x_1}, \ldots, D_{x_{i-1}}, [a, a^+), D_{x_{i+1}}, \ldots, D_{x_k})$,
2. $C(D_{x_1}, \ldots, D_{x_{i-1}}, (b^-, b], D_{x_{i+1}}, \ldots, D_{x_k})$.
where a^+ (resp. a^-) corresponds to the smallest (resp. largest) number of $\overline{\mathbb{F}}$ strictly greater (resp. smaller) than a, and C stands for interval extension of c [3].

The essential point is that the variable x is Box–consistent for constraint $f(x, x_1, \ldots, x_n) = 0$ if the bounds of the domain of x correspond to the leftmost and the rightmost 0 of the optimal interval extension of $f(x, x_1, \ldots, x_n)$.

[4]It is worthwhile to notice that the set of constraints $\mathcal{C}$ represents a **conjunction** of constraints that have to be satisfied. Disjunctions may only occur inside a single constraint, e.g. the single constraint $x^2 = y$ is equivalent to the disjunction $(x = \sqrt{y}) \vee (-x = \sqrt{y})$.

3 Extension of the domain of a variable of a CSP

This section introduces the way a domain of a single variable can be extended while preserving consistency of the whole CSP. We start by defining two local consistencies which are needed to characterize the extended domains.

Next, we formally define the univariate extrema functions that actually compute the bounds of the i–consistent extensions of the domain of a variable.

3.1 e–consistency

Various approximations of arc consistency (e.g. 2B–consistency[7], Box–consistency[1]) have been introduced for continuous domains. e–consistency is the best approximation of the solution space which can be computed by these partial consistencies. For instance, e–consistency corresponds to the "smallest external box" on Fig. 2. More formally, e–consistency is defined as follows:

Definition 7 (e–consistency) *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be an arc consistent CSP. A CSP $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is e–consistent iff* $\forall D'_i \in \mathcal{D}' : D'_i =$`hull`$(D_i)$

In other words, a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is e–consistent iff $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is arc consistent and $\mathcal{D}$ corresponds to the smallest box containing all values of $\mathcal{D}'$. So, for inequality c, e–consistency on the corresponding equation c_{equ} (see section 3.4) yields a box which bounds the maximal extension that can be performed for any variable occurring in c.

3.2 i–consistency

Definition 8 (i–consistency)
Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. P is i–consistent iff $\forall c \in \mathcal{C}, \forall \vec{v} \in \vec{\mathcal{D}}$: $c(\vec{v})$

In other words, a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is i–consistent iff $\mathcal{D}$ only contains tuples which are solutions.

Example 2 *Let P be the CSP defined by $\mathcal{X} = \{x, y\}$, $D_x = [-2, 2], D_y = [6, 12]$, $\mathcal{C} = \{x^2 - y \leq 0, -x^2 + 20 \leq y\}$. This system is i–consistent :*
$\forall (v_x, v_y) \in (D_x, D_y)$: $v_x^2 - v_y \leq 0 \wedge -v_x^2 + 20 \leq v_y$
Now, we want to find the largest $D'_x \supseteq D_x$ such that the CSP defined by replacing D_x by D'_x in P is i–consistent.

Figure 3 shows the original box and the extension of D_x. Both boxes are i–consistent[5]*. The domain of y remains unchanged. B is the e–consistent box for equation $x^2 = y$ and gives the upper bound of the extension of D_x.*

Ward et al. [13] have proposed four kinds of interval propagation. One of them is related to i–consistency. Each interval D_x is labeled with one of these kinds:

[5]Note that we could also perform a fruitful i–consistent extension of D_y to $[6, 14]$ with the new box. But this extension of D_y is much smaller than the one we would have obtained if we had extended the initial box (D_y would have been extended to $[4, 16]$). In general, the result of successive extensions by i–consistency of several variables depends on the processing order of the variables.

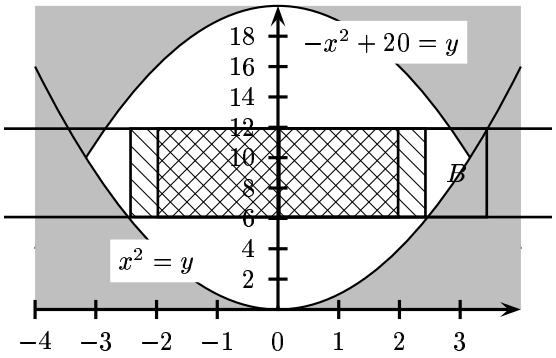


Figure 3: Maximal i–consistent extension of D_x

"only", "every", "some" and "none". If D_x is labeled "only" then solution tuples only take their values for x in D_x. If D_x is "every" then every value of x in D_x gives a tuple solution. If D_x is "some" then there exists at least one solution tuple such that x takes its value in D_x. If D_x is "none" then there is no solution tuple such that the value of x is in D_x.

Labelling every variable with "every" is what we call i–consistency. However, Ward et al.'s inference rules that allow computing labelled interval propagation do not consider the case where two variables are labelled "every". Moreover, these inference rules assume strong monotony and continuity properties of the constraint system.

Now, we formally define what we mean by right i–consistent extension.

3.3 Right i–consistent Extension of D_x

Definition 9 (Right i–consistent extension of D_x)

Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a i–consistent CSP. $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is a right i–consistent extension of D_x for P iff:

- $\forall D_i \in \mathcal{D} \setminus \{D_x\},\ D_i = D'_i$
- $D_x \subset D'_x,\ \underline{D'_x} = \underline{D_x}$
- *P' is i–consistent*

Definition 10 (Maximal right extension)
Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a i–consistent CSP. $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ is a maximal right i–consistent extension of D_x for P iff:

- *P' is a right i–consistent extension of D_x for P;*
- *$\forall P''$, such that P'' is a right i–consistent extension of D_x for P, $P'' \subset P'$* [6]

3.4 Extrema functions

Let c be an inequality, c_{equ} denotes the equation corresponding to c. More precisely, if c is defined by an expression of the form $f(x_1, \ldots, x_n) \leq 0$ or $f(x_1, \ldots, x_n) \geq 0$, then c_{equ} denotes the equation $f(x_1, \ldots, x_n) = 0$.

[6] A CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is smaller than a CSP $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ if $\mathcal{D} \subseteq \mathcal{D}'$. $\mathcal{D} \subseteq \mathcal{D}'$ means $D_{x_i} \subseteq D_{x'_i}$ for all $i \in 1..n$.

$F_c^{min(x)}(\mathcal{D})$ is an optimal extremum function of c_{equ} for variable x if $F_c^{min(x)}(\mathcal{D})$ computes the smallest value of x which is a solution of c_{equ} [7] in the space delimited by $\mathcal{D}$.

Definition 11 (Optimal extremum function)
Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP, x a variable of $\mathcal{X}$ and c a constraint of $\mathcal{C}$. The optimal extremum function of constraint c_{equ} for variable x is

$F_c^{min(x)}(\mathcal{D}) = min(\pi_x(\vec{v}) \mid c_{equ}(\vec{v})$ *holds.)*

By convention, $F_c^{min(x)}(\mathcal{D})$ returns $\overline{D_x}$ when c_{equ} has no solution in the space delimited by $\mathcal{D}$.

Definition 12 (Extremum function approximation)

Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP, c a constraint of $\mathcal{C}$ and $F_c^{min(x)}(\mathcal{D})$ an optimal extremum function of c_{equ} for variable x.
$AF_c^{min(x)}(\mathcal{D})$ is a safe approximation of $F_c^{min(x)}(\mathcal{D})$ iff:
$AF_c^{min(x)}(\mathcal{D}) < F_c^{min(x)}(\mathcal{D})$

3.5 Computing an i–consistent right extension of D_x for P

To define the right i–consistent extension of D_x for a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, we introduce two specific domains, named $\mathcal{D}_{max}$ and $\mathcal{D}_{ext}$:

- $\mathcal{D}_{max}$ is the set of initial domains where D_x has been set to $[\overline{D_x}, \infty)$, i.e., $\mathcal{D}_{max} = \mathcal{D}_{D_x \leftarrow [\overline{D_x}, \infty)}$
- $\mathcal{D}_{ext}$ is the set of initial domains where D_x has been extended to the value of the extremum function of constraint c on x,
  i.e., $\mathcal{D}_{ext} = \mathcal{D}_{D_x \leftarrow [\underline{D_x}, F_c^{min(x)}(\mathcal{D}_{max})]}$

Next proposition defines the right i–consistent extension of D_x for a CSP P with only one single constraint.

Proposition 1 *Let $P = (\mathcal{X}, \mathcal{D}, \{c\})$ be an i–consistent CSP with only one inequality constraint c and let x be a variable of $\mathcal{X}$. Then, $P = (\mathcal{X}, \mathcal{D}_{ext}, \{c\})$ is a maximal right i–consistent extension of D_x for P.*

Proof:
We assume that c is not a tautology [8] which is true for any value of x. So it results from the definition of the extrema functions (definition 9) that we have either:

$\forall \vec{v} \in \mathcal{D}_{ext}\ :\ c(\vec{v})$

or $\forall \vec{v} \in \mathcal{D}_{ext}\ :\ not(c(\vec{v}))$

Since P is i–consistent, it results that $\forall \vec{v} \in \mathcal{D}_{ext}\ :\ c(\vec{v})$.

Proposition 2 *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a i–consistent CSP. Let $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C})$ such that:*

- $\forall D_i \in \mathcal{D} \setminus \{D_x\} : D_i = D'_i$

[7] Of course, when c_{equ} is not defined in a subpart of $\mathcal{D}$, $F_c^{min(x)}(\mathcal{D})$ returns a value that is strictly smaller than the smallest value of D_x for which c_{equ} is not defined.

[8] Tautologies can be removed in a pre-processing step.

- $D'_x = [\underline{D_x}, min_{c \in C}(F_c^{min(x)}(\mathcal{D}_{max}))]$

Then, P' is a right i–consistent extension on x of P. Furthermore, P' is a maximal right i–consistent extension of P on x.

Proof: From proposition 1, it results that $\forall \vec{v} \in D', \forall c \in C : c(\vec{v})$. Since P is a conjunction of constraints, P' is i–consistent.
Let $P'' = (\mathcal{X}, \mathcal{D}'', \mathcal{C})$ be an i–consistent extension of D_x for $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. Assume that $P' \subset P''$ and that:

- $\forall D''_i \in \mathcal{D}'' \setminus \{D''_x\} : D'_i = D''_i$
- $D'_x \subset D''_x$
- $\underline{D''_x} = \underline{D'_x} = \underline{D_x}$

Assume that c^k is the constraint such that $\overline{D'_x} = F_{c^k}^{min(x)}(\mathcal{D}_{max})$. So, there exists $\vec{v} \in \mathcal{D}_{max}$ such that c^k is false, and that $\overline{D'_x} < \Pi_x(\vec{v}) \leq \overline{D''_x}$. Thus, P'' is not i–consistent.

The algorithm in Figure 4 directly follows from property 2. Note that this algorithm is much simpler than the framework introduced by Sam-Haroud and Faltings [9] to compute a local consistency. Both algorithms select relevant extrema from all extrema including intersections between several curves and intersections between curves and interval extremities. However, in our case, the relevant extrema is simply the left most one since we start from an initial i–consistent box (so we know which portion of the space is a solution) and we extend only one variable domain to the right. This algorithm only searches for the left most extrema, thus, it is linear if the extrema functions can be computed in constant time. The next section shows that the left most extrema can be computed very efficiently.

```
function i-extension(x,D_max,C): real
Ī ← Max-Value
for c in C
        Ī ← Min(Ī, F_c^min(x)(D_max))
return Ī
End function
```

Figure 4: function `i-extension`

3.6 Computing extrema functions

Optimal extrema functions for variable x of constraint c can trivially be computed if c is either a monotonic on x, or if D_x can be decomposed in subdomains where c is monotonic on x. Such constraints are usually called primitive constraints[3]. The set of primitive constraints is infinite and includes the following constraints: $\{x = y, x \leq y, x < y, x \neq y, z = x + y, z = x * y, x = -y, y = sin(x), y = cos(x), y = e^x, y = abs(x), z = x^y, \ldots\}$.

Example 3 *The constraint $x^3 = y$ is primitive: the right extrema function for x is :*

$$F_x^{min}(D_x, D_y) = max(\underline{D_x}, \sqrt[3]{\underline{D_y}}))$$

For a non–primitive constraint c, we will approximate the e–consistent box for c_{equ} in the space delimited by domains $\mathcal{D}_{D_x \leftarrow D_{max}}$. The methods introduced to compute Box–consistency provide an efficient way to compute such a safe approximation of $F_c^{min(x)}(\mathcal{D}_{max})$. The key observation is that extrema functions are *univariate functions* which can be tackled by the *Newton method* implemented in the Box–consistency.

So, consider the i–consistent extension of D_x for CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and an inequality $c \in \mathcal{C}$. To compute a safe approximation of the extrema functions for x of constraint c, we could just compute a Box–consistent interval for x with regard to c_{equ}. Box–consistency would yield an interval D'_x such that $\Pi_x(sol(c_{equ}, \mathcal{D}_{max})) \subset D'_x$. Thus, $\underline{D_x} = AF_c^{min(x)}(\mathcal{D}_{max})$.

As a matter of fact, a complete computation of Box–consistency is not required. The `LNAR` procedure [11] used in Box–consistency finds the left most zero of the interval extension of the univariate function on x derived from c_{equ} by replacing all variables but x by their domains. Of course, when the function `i-extension` (see. fig. 4) uses approximations of extrema functions, the i–extension of the domain of x may not be maximal.

4 A ballistic example

In this section, we give a small ballistic application which illustrates the capabilities of our system. The problem consists of finding the maximum mechanical tolerances when an object is launched in a uniform gravitational field $\vec{g}$, with an initial speed $\vec{V_i}$ which has an incidence α with the ground (see fig. 5).

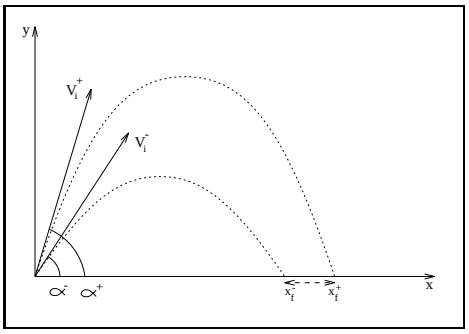


Figure 5: Possible trajectories of the projectile

The strong requirement is that the object must fall inside a predefined interval.

4.1 Modeling of the problem

The initial speed and incidence of the bullet can be stated as follows:

$x_f = V_x t_f \qquad y_f = -\frac{1}{2} g t_f^2 + V_y t_f$
$V_x = V \cos(\alpha) \qquad V_y = V \sin(\alpha)$

These equations give the impact point (x_f, y_f) of the bullet when $t = t_f$ and $y_f = 0 : t = V_y \frac{2}{g}$
Thus, $x = V_x \frac{2V_y}{g} = \frac{2V^2}{g} \cos(\alpha) \sin(\alpha)$

4.2 Computing i–consistency extension of D_α

The target is defined by the interval $[220, 250]$. Now, assume that the bullet falls on the target when $\alpha \in [32, 35]$ and $V \in [49.2, 50.1]$. So the initial i–consistent box is defined by :
$\forall \alpha \in D_\alpha$:

$\frac{2}{9.81} * [49.2, 50.1]^2 * cos(\alpha) * sin(\alpha) \geq 220$ (c)

$\frac{2}{9.81} * [49.2, 50.1]^2 * cos(\alpha) * sin(\alpha) \leq 250$ (c')

To extend D_α to the right by i–consistency, we have to check whether the box at the right of the i–consistent box is i–consistent. Thus, we have to find the left most bound of D_α for c_{equ} and c'_{equ} with $D_{max} = [35, 90]$. To find these bounds, we have used `Numerica` [10] to compute Box–consistent intervals for c_{equ} and c'_{equ}. The left most bound of these intervals respectively are 58.4 and 38.4. So, D_α can be extended by i–consistency to interval $[32, 38.4]$ (see Fig. 6).

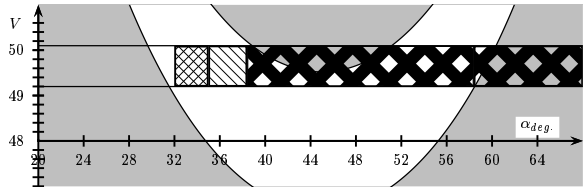


Figure 6: The ballistic constraints and boxes

Now, consider the three dimensional version of this problem where the target is defined by a rectangular area of space $R_x \times R_y \times R_z$ such that $220 \leq R_x \leq 250$, $R_y = 0$ and $-50 \leq R_z \leq 50$. Let β be the angle between $\vec{x}$ and the projection of $\vec{V}$ on the plane defined by $y = 0$. We know that the box defined by $D_\alpha = [34, 35]$,$D_V = [49.2, 50.1]$ and $D_\beta = [0, 0]$ is i–consistent.
To extend β by i–consistency, we have computed with `Numerica` the Box–consistent intervals for the equations derived from the following inequalities :
$220 \leq \frac{2}{9.81} * V^2 * sin(\alpha) * cos(\alpha) * cos(\beta)$
$250 \geq \frac{2}{9.81} * V^2 * sin(\alpha) * cos(\alpha) * cos(\beta)$
$-50 \leq \frac{2}{9.81} * V^2 * sin(\alpha) * cos(\alpha) * sin(\beta)$
$50 \geq \frac{2}{9.81} * V^2 * sin(\alpha) * cos(\alpha) * sin(\beta)$
The left most bound of these intervals is 1.4; thus, D_β can be extended by i–consistency to the interval $[0, 1.4]$.

5 Conclusion

This paper has introduced an effective framework for extending the domain of one variable in an already consistent CSP. Extending the domain of one variable is a critical issue in applications where the tolerance of a component determines its cost.
Contrary to Ward et al [13] we do not impose any restrictions on the form of the constraints. The approach suggested by Sam-Haroud and Faltings [9] is more general since they do not know an initial solution but its computation cost is very high. The key point of our framework is the definition of univariate extrema functions which can be computed efficiently.
An interesting way to explore concerns maximizing the size (or volume) of i–consistent boxes.

Acknowledgements

Thanks to Gilles Trombettoni for his careful reading and helpful comments on earlier drafts of this paper. Thanks also to Olivier Lhomme, Jean-Paul Stromboni and Alexander Semenov for interesting suggestions.

References

[1] F. Benhamou, D. McAllester and P. Van Hentenryck. CLP(intervals) revisited. In *Proc. ILPS*. MIT Press, 1994.

[2] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1997.

[3] H. Collavizza, F. Delobel and M Rueher. A note on partial consistencies over continuous domains. In *Proc. of CP 98*, LNCS 1520, Springer Verlag, 1998.

[4] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[5] B. Faltings. Arc–consistency for continuous variables. *Artificial Intelligence*, 65:363–376, 1994.

[6] B. Faltings and E. Gelle. Local consistency for ternary numeric constraints. *IJCAI'97*, pages 392–397, 1997.

[7] O. Lhomme. Consistency techniques for numeric csp. *Proc. IJCAI93*, pp. 232–238, 1993.

[8] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):pp. 99–118, 1977.

[9] D.J Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1& 2):85–118, September 1996.

[10] P. Van Hentenryck, Y. Deville and L. Michel. *Numerica. A modeling language for global optimization.* MIT Press, 1997.

[11] D. Kapur, D. McAllester and P. Van Hentenryck. *Solving Polynomial Systems Using a Branch and Prune Approach SIAM Journal on Numerical Analysis*, 34(2), April 1997.

[12] D. Waltz. *The Psychology of Computer Vision*, chap. Understanding line drawing of scenes with shadow. McGraw-Hill, 1975.

[13] A.C. Ward, T. Lozano-Perez and W.P. Seering. Extending the constraint propagation of intervals. In *Proc. of the 11th IJCAI*, pages 1453–1458, 1989.

Bibliographie

[1] I. Araya, B. Neveu et G. Trombettoni : An interval constraint propagation algorithm exploiting monotonicity. *In* Springer Verlag, éditeur : *International workshop IntCP at CP conference*, numéro 5732 de LNCS, pages 65–83, 2009.

[2] F. Benhamou, D. Mc Allester et P. Van Hentenryck : CLP(Intervals) revisited. *In International Symposium on Logic Programming*. MIT Press, 1994.

[3] F. Benhamou et F. Goualard : Universally quantified interval constraints. *In Procs. of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'2000)*, volume 1894 de *Lecture Notes in Computer Science*, pages 67–82, Singapore, 2000. Springer-Verlag.

[4] F. Benhamou, F. Goualard et L. Granvilliers : Programming with the DecLIC language. *In Second workshop on Interval Constraints,Port-Jefferson, NY, USA*, October 1997.

[5] F. Benhamou, F. Goualard, E. Languénou et M. Christie : Interval constraint solving for camera control and motion planning. *ACM Transactions on Computational Logic*, 5(4), 2004.

[6] F. Benhamou et W. Older : Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.

[7] B. Botella, A. Gotlieb et C. Michel : Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):97–121, 2006.

[8] G. Chabert et L. Jaulin : Contractor programming. *Artifical Intelligence*, 173:1079–1100, 2009.

[9] G. Chabert, L. Jaulin, B. Neveu et G. Trombettoni : Ibex - an interval based explorer. `http://www.ibex-lib.org/`.

[10] J.C. Cleary : Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[11] H. Collavizza, F. Delobel et M. Rueher : A note on partial consistencies over continuous domains solving techniques. *In Fourth International Conference on Principles and Practice of Constraint Programming*, numéro 1520 de LNCS. Springer Verlag, 1998.

[12] H. Collavizza, F. Delobel et M. Rueher : Comparing partial consistencies. *Reliable Computing*, 5(3):213–228, 1999.

[13] H. Collavizza, F. Delobel et M. Rueher : Extending consistent domains of numeric CSP. *In IJCAI'99*, pages 406–411. Morgan Kaufmann, 1999.

[14] Rina Dechter : *Constraint Processing*. Morgan Kaufmann, 2003.

[15] F. Delobel : *Résolution de systèmes de contraintes réelles non linéaires*. Thèse de doctorat, Université de Nice Sophia Antipolis, Sophia Antipolis, France, 2000.

[16] B. Faltings : Arc–consistency for continuous variables. *Artificial Intelligence*, 65:363–376, 1994.

[17] E. C. Freuder : Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, November 1978.

[18] E. Gardeñes, M. Á. Sainz, L. Jorba, R. Calm, R. Estela, H. Mielgo et A. Trepat : Modal intervals. *Reliable Computing*, 7(2):77–11, 2001.

[19] A. Goldsztejn : *Définition et Applications des Extensions des Fonctions Réelles aux Intervalles Généralisés*. Thèse de doctorat, Université de Nice Sophia Antipolis, Sophia Antipolis, France, 2005.

[20] C. Grandón, G. Chabert et B. Neveu : Generalized interval projection : A new technique for consistent domain extension. *In 20th IJCAI*, pages 94–99, 2007.

[21] L. Granvilliers et F. Benhamou : Algorithm 852 : Realpaver : An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1):138–156, 2006.

[22] E. Hansen : *Global optimization using interval analysis*. Marcel Dekker, NY, 1992.

[23] P. Van Hentenryck, Y. Deville et L. Michel : *Numerica. A modeling language for global optimization*. MIT Press, 1997.

[24] P. Van Hentenryck, D. McAllester et D. Kapur : Solving polynomial systems using a branch and prune aprroach. *SIAM Journal on Numerical Analysis*, 34(2), April 1997.

[25] H. Hong et V. Stahl : Safe starting regions by fixed points and tightening. *Computing*, 53:323–335, 1994.

[26] R. Baker Kearfott : *Rigorous Global Search : Continuous Problems*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1996.

[27] R. Baker Kearfott : Globsol : History, composition, and advice on use. *In* Springer Verlag, éditeur : *CPAIOR*, numéro 2861 de LNCS, pages 17–31, 2003.

[28] Yahia Lebbah : Icos : Interval constraints solver. `http://ylebbah.googlepages.com/icos`.

[29] Yahia Lebbah, Claude Michel et Michel Rueher : A rigorous global filtering algorithm for quadratic constraints. *CONSTRAINTS Journal*, 10(1), 2005.

[30] J. H. M. Lee et M. H. van Emden : Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16(3–4):255–276, 1993.

[31] Bruno Legeard, Michel Rueher, Thierry Jéron et Bruno Marre : Rapport final du projet ACI V3F : Validation et vérification en présence de calculs en virgule flottante. `http://lifc.univ-fcomte.fr/v3f/rapport/rapporttechnique/rapportcomplet/22_rapporttechnique_complet_ACISI_V3F-final.pdf`.

[32] O. Lhomme : Consistency techniques for numeric CSPs. *In* Ruzena Bajcsy, éditeur : *IJCAI 1993*, pages 232–238. Morgan Kaufmann, 1993.

[33] O. Lhomme et M. Rueher : Application des techniques CSP au raisonnement sur les intervalles. *RIA*, 11(3):283–312, 1997.

[34] A. Mackworth : Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[35] Jean-Pierre Merlet : ALIAS-C++- a C++ algorithms library of interval analysis for equation systems. `http://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS-C++/ALIAS-C++.html`.

[36] C. Michel, M. Rueher et Y. Lebbah : Solving constraints over floating-point numbers. *In* Springer Verlag, éditeur : *CP'2001*, numéro 2239 de LNCS, pages 524–538, 2001.

[37] R. Moore : *Interval Analysis*. Prentice Hall, 1966.

[38] A. Neumaier : *Interval methods for systems of equations*. Cambridge University Press, 1990.

[39] A. Neumaier, O. Shcherbina, W. Huyer et T. Vinkó : A comparison of complete global optimization solvers. *Mathematical Programming*, 103(2):335–356, 2005.

[40] Parc technologique de Luminy Case 919 13288 Marseille cedex 09 (France) Prologia : PrologIV constraints inside, 1996.

[41] J-F. Puget et P. Van Hentenryck : A constraint satisfaction approach to a circuit design problem. *Journal of Global Optimization*, 13:75–93, 1998.

[42] Michel Rueher : Cours programmation par contraintes (master II ISI et PLMT), département SI de polytech nice-sophia. `http://users.polytech.unice.fr/~rueher/teaching.html`, intervalles 2.

[43] D.J Sam-Haroud et B. Faltings : Consistency techniques for continuous constraints. *Constraints*, 1(1& 2):85–118, September 1996.

[44] E. Tsang : *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[45] A.C. Ward, T. Lozano-Perez et W.P. Seering : Extending the constraint propagation of intervals. *In 11th IJCAI*, pages 1453–1458, 1989.

Chapitre VI

Travaux en cours et perspectives

Ce chapitre présente les perspectives de mes travaux. D'une part, je décris les travaux que nous effectuons en collaboration avec Michel Rueher, avec lequel je co-encadre la thèse de doctorat de Le Vinh Nguyen, sur le thème de la vérification des programmes temps réel. Le défi de Le Vinh est d'appliquer cette approche à des problèmes réels, fournis par nos partenaires industriels du projet TESTEC «TEst des Systèmes Temps réel Embarqués Critiques». D'autre part, je présente les perspectives de mes travaux sur la vérification formelle des programmes avec HOL.

VI.1 Vérification formelle de programmes temps réel

Les travaux sur la vérification formelle de programmes temps réel font suite à l'approche par exécution symbolique présentée section III.4. Je critique donc tout d'abord cette approche pour motiver les nouveaux choix qui ont été faits. Je décris ensuite un cas d'étude fourni par nos partenaires industriels du projet TESTEC qui a mis en échec cette première approche et soulève de nombreuses perspectives de recherche. Enfin, je termine par les travaux futurs, en particulier le traitement des programmes avec des flottants.

VI.1.1 Critique de l'approche par exécution symbolique avec les contraintes

L'approche présentée section III.4 a été implémentée en Java en utilisant les solveurs d'Ilog *Jsolver* et MIP. Ces choix ont été guidés par le souci d'implémenter rapidement un premier prototype. J'ai donc choisi le langage de programmation qui m'est actuellement le plus familier, et les solveurs de contraintes pour lesquels nous avions un support fiable, puisque nous entretenons une collaboration avec l'équipe CP d'Ilog implantée à Sophia Antipolis. Cependant, cette première implémentation n'est pas du tout optimisée et a les défauts suivants :

- La gestion mémoire en Java rend l'exécution des programmes récursifs très lente.
- L'exploration des différents chemins du programme est gérée en profondeur d'abord sans aucune optimisation ou parallélisation.
- La résolution des systèmes de contraintes est faite avec les paramètres par défaut et sans avoir écrit de statégie de recherche spécifique.
- Si tout est écrit de façon uniforme en Java (Jsolver est une bibliothèque Java et j'ai utilisé une interface Java du MIP qui est lui-même écrit en C++), les structures de données ne sont pas partagées.
- Enfin, les produits d'Ilog sont commerciaux ce qui rend impossible la diffusion de notre outil de vérification de programmes. Ceci est préjudiciable quand on souhaite se comparer à d'autres approches qui elles fournissent un outil libre.

Nous avons donc proposé à Le Vinh d'explorer les possibilités offertes par le solveur de contraintes COMET [2] qui offre les atouts suivants :

- Il est libre pour les académiques.
- COMET est un langage de programmation par contraintes qui offre à la fois les paradigmes d'un langage de programmation objet et les concepts clefs de la résolution de contraintes : définir un modèle et chercher des solutions.

– COMET offre de nombreuses possibilités pour explorer le système de contraintes de façon efficace comme par exemple un mécanisme de continuation et de retour arrière fort avantageux pour explorer les différentes branches d'une instruction conditionnelle.
– Enfin, COMET offre plusieurs "moteurs" de résolution comme le cadre CP ou un solveur linéaire ; il fournit aussi une API pour intégrer un solveur sur les flottants.

La première année de la thèse de Le Vinh a consisté à comprendre le problème et à retranscrire la version développée en Java dans COMET. Il a maintenant acquis une bonne compréhension des paradigmes de COMET et a implémenté plusieurs versions (en utilisant la recherche non déterministe) qui montrent une nette amélioration des performances par rapport à l'approche en Java. Je ne développe pas ces résultats qui sont le cœur de la thèse de Le Vinh. J'ai choisi par contre de présenter un cas d'étude qui a mis en échec l'approche par exécution symbolique et qui rend pertinente la réflexion sur l'avantage des preuves en avant comparée à celle des preuves en arrière.

VI.1.2 Gestionnaire de clignotants : un cas d'étude

Ce cas d'étude a été fourni par la société *Geensys* partenaire industriel du projet TESTEC. Il s'agit d'un gestionnaire de clignotants dont nous possédons une description *Simulink*. Ce gestionnaire a été testé par *Geensys* avec des outils propriétaires qui ont également permis d'en générer un programme C (i.e une fonction qui décrit les sorties en fonction des entrées). Cet exemple est de taille importante (en tout cas par rapport aux exemples traités précédemment). En effet, il contient 80 variables booléennes, 28 variables entières, et a une structure de contrôle complexe (de nombreuses conditionnelles emboîtées, de l'ordre de 500 lignes de code).

Une des propriétés qui doit être vérifiée pour ce gestionnaire de clignotants est que la sortie, c'est à dire l'ampoule du clignotant, ne doit pas rester tout le temps allumée, et ceci quelle que soit la fonctionnalité invoquée (e.g. feux de détresse, signalement d'ouverture de porte ou clignotants). Le programme C a donc été modifié par Thierry Gueguen de *Geensys* pour compter le nombre de fois consécutives où la sortie a la valeur 1 (il suffit pour cela d'écrire une boucle qui appelle la fonction C de base et de gérer un compteur qui est incrémenté si la sortie est à 1 et qui repasse à 0 si la sortie vaut 0). Le programme est correct pour n passages dans la boucle si la valeur du compteur est strictement inférieure à $n-1$ à la sortie de la boucle ; ceci est inséré comme une assertion. Si cette assertion est violée, alors le clignotant est resté allumé pendant n étapes consécutives et la propriété est donc violée.

Le programme C qui gère le compteur et l'assertion a pu être vérifié avec CBMC (voir section III.4.5) jusqu'à 150 passages dans la boucle en 63.64s. Avec notre approche précédente, implémentée de façon naïve en COMET (i.e appel au solveur chaque fois qu'un test est atteint) nous n'avons pas pu vérifier d'instance plus grande que 4 (trop de conditions à tester). En effet, une caractéristique importante de ce programme est qu'il contient une erreur : pour tous les n testés, nous avons trouvé une configuration des entrées telle que la sortie restait allumée pendant n itérations. Notre approche teste successivement tous les chemins. Ceci peut être un défaut quand le programme contient une erreur car le chemin qui la contient peut être

testé en dernier. Par contre, CBMC construit une formule booléenne qui représente tous les chemins (i.e grosse formule conditionnelle qui contient à chaque fois les deux branches d'une instruction conditionnelle). Cette formule toute entière est fournie en entrée d'un solveur SAT. La différence fondamentale est donc que le solveur SAT a une vision globale du système alors que notre approche en profondeur d'abord n'a qu'une vision séquentielle des instructions du programme. Dans le cas où le programme est correct (cas que nous avons traité le plus souvent dans les exemples précédents), cela n'a pas d'incidence puisque de toute façon, tous les chemins doivent être explorés. Par contre, cette vision partielle n'est pas adaptée pour identifier au plus vite un chemin erroné.

VI.1.3 Travaux futurs

Parcours en largeur d'abord La faiblesse de notre approche sur le gestionnaire de clignotants amène une première réflexion : un parcours en profondeur d'abord est-il bien adapté ? Ceci est certainement avantageux dans le cas où la pré-condition et les décisions prises sur le chemin courant contraignent fortement les variables du programme (e.g. recherche binaire dans un tableau ordonné ou programme *Tritype* où de nombreux tests portent sur des variables locales). Mais en présence d'erreur, la connaissance restreinte que l'on a du programme peut retarder la détection de cette erreur. Une première piste serait d'effectuer un parcours en largeur d'abord. Ceci pourrait être implémenté efficacement grâce au parallélisme dans COMET : pour chaque instruction conditionnelle, les deux branches seraient testées en parallèle, le processus s'arrêtant dès qu'une erreur a été trouvée ou quand toutes les branches ont été explorées.

Une idée orthogonale, proche des réflexions de la section VI.2, pourrait être d'explorer le graphe de flot de contrôle "à l'envers", en partant de la post-condition, ou même de définir des stratégies pour travailler à la fois en profondeur d'abord et en partant de la post-condition.

Cas des programmes non linéaires Un autre travail à effectuer concerne le cas des programmes qui contiennent des expressions non linéaires. En effet, nous devons vérifier dans le cadre du projet TESTEC un deuxième cas d'étude fourni par *Geensys* qui modélise un système de frein ABS. Dans ce système, le taux de glissement du véhicule est défini par $t_g = 1 - w \times r/v$ à partir de sa vitesse de translation v, du rayon des roues r et de la vitesse de rotation des roues w. L'anti-blocage est invoqué quand la vitesse de la roue devient inférieure à $(1 - t_g)/v_{ref}$ où v_{ref} est une vitesse de référence calculée à partir de la vitesse de rotation de deux roues prises en diagonale. Ce système contient des expressions non linéaires, il faut donc un solveur spécifique pour le résoudre. Le choix de ce solveur sera effectué en fonction de l'objectif fixé pour la vérification. Si nous choisissons de vérifier les propriétés physiques du système ABS, le raisonnement doit se faire sur les réels, et nous utiliserons un des solveurs sur les réels présentés dans la discusion du chapitre V (section V.4 page 228). Par contre, comme c'est probable, si nous continuons dans l'optique de vérifier un programme C généré à partir d'une description *Simulink*, l'idée sera plutôt de vérifier que ce programme répond à ses spécifications. Dans ce

cas, il s'agira de prouver que le programme calcule ce qui est attendu, mais ceci sur les flottants, puisque c'est ce que manipulent les programmes C. Nous utiliserons donc le solveur de contraintes développé par Claude Michel dans le cadre du projet $V3F$ (voir discussion section V.4). De façon pratique, dans les deux cas, nous pourrons utiliser l'API fournie par COMET pour intégrer le solveur non-linéaire.

Notons que lors de l'exécution symbolique du programme, les tests de conditions non linéaires pourraient faire appel à un calcul d'approximation comme celui du solveur $ICOS$ (voir section V.4). En effet, si la condition est quadratique, calculer une approximation linéaire et appeler un solveur linéaire (donc efficace) permettrait de prendre une décision dans le cas où la relaxation linéaire est insatisfiable, puisque dans ce cas la condition non linéaire l'est aussi.

Détection des erreurs de débordement Un aspect que je n'ai pas du tout traité, mais qui mettrait sûrement à profit le mécanisme de propagation des contraintes, est la détection de conditions d'erreur telles que les divisions par zéro ou la sortie d'un indice de parcours d'un tableau. En effet, pour un tableau de longueur l, vérifier qu'un indice i ne sort pas des bornes du tableau pourrait être effectué en construisant le CSP où le domaine initial de i est $[l+1, max_{int}]$, et les contraintes sont issues des instructions qui modifient i, (ces contraintes seraient construites en parcourant le programme "en arrière", en partant de la dernière affectation de i). Dans de nombreux cas, un simple filtrage permettrait de conclure que ce CSP ne contient pas de solution et donc que $i < l$.

Aide à la correction du programme Enfin, un dernier aspect à explorer concerne l'aide à la correction des programmes en cas d'erreur. En effet, le mécanisme de recherche de solutions est bien adapté pour fournir plusieurs valeurs numériques pour chaque cas d'erreur. Ceci s'oppose aux outils de BMC basés sur des solveurs SAT qui ne renvoient qu'une trace du chemin erroné.

VI.2 Travaux sur la vérification des programmes basée sur la sémantique

Je compte aussi poursuivre mes travaux sur la vérification des programmes basée sur la sémantique. En effet, cette approche plus formelle me paraît complémentaire à l'approche basée sur la programmation par contraintes. Bien que souvent moins efficace, l'écriture rigoureuse en HOL aide à formaliser le mécanisme de la preuve, et fournit un cadre sémantique cohérent pour évaluer aussi bien les preuves en avant (complètes ou par BMC) que les preuves en arrière. De plus, combiner une preuve automatique et une preuve mécanique dans un assistant de preuve est un problème fondamental et difficile mais qui peut être utile en pratique. Le cas du gestionnaire de clignotants de la section précédente en est une bonne illustration. En effet, avec les outils de BMC, nous avons toujours pu trouver une erreur dans le programme, et ce quel que soit le nombre de passages dans la boucle que nous avons testé. Il est donc naturel, pour corriger l'erreur, de chercher à savoir si une telle erreur existe

pour tout passage dans la boucle. Ceci peut être formalisé en HOL et pourrait être prouvé mécaniquement (avec un effort important bien sûr).

Nous avons formalisé section IV.3 le bounded model checking par exécution symbolique comme une génération de post-conditions. Une première voie que nous voulons explorer est l'utilisation des règles de génération d'obligations de preuve en avant, et ce en pur HOL. L'évaluation des conditions sur l'état courant et la simplification du terme obtenu pourrait être rendue efficace par l'appel à un solveur externe, comme en particulier le solveur SMT *Yices* qui est actuellement en cours d'intégration à HOL4. L'idée est d'évaluer l'efficacité de telles preuves par rapport aux preuves en arrière généralement faites par les outils qui intègrent des démonstrateurs de théorèmes, et éventuellement de combiner les deux approches.

Une autre idée qui nous semble importante est l'association d'une preuve complète dans le cas où l'invariant est fourni, avec une preuve par BMC dans le cas contraire[1]. Le principe pourrait être de déplier les boucles sans invariant dans une phase de pré-traitement et d'effectuer ensuite une preuve complète en HOL du programme obtenu. Ce pré-traitement peut être écrit en HOL pur et il semble envisageable de montrer l'équivalence entre la boucle dépliée et la boucle initiale exécutée un certain nombre de fois (en gérant comme dans les outils de BMC une *assertion de boucle* qui est violée si le nombre de dépliage est insuffisant).

Les deux points précédents supposent qu'HOL4 soit assez efficace pour pouvoir appliquer la méthode à des exemples significatifs. Ceci n'est pas une certitude, et une version efficace écrite en ML et compilée pourrait être dérivée directement des définitions et théorèmes de la version HOL pure. L'idée serait de pouvoir "basculer" de la preuve automatique en ML vers une preuve interactive en HOL dans le cas où les obligations de preuve n'ont pas toutes été déchargées.

Enfin, il semble aussi intéressant, de façon orthogonale aux travaux ci-dessus, de réfléchir à l'intégration d'invariants de boucle dans notre approche de BMC par programmation par contraintes.

[1] Une autre possibilité est la génération automatique d'invariants comme dans [1].

Bibliographie

[1] Willem Visser Corina S. Pasareanu : Verification of java programs using symbolic execution and invariant generation. *In SPIN*, volume 2989 de *LNCS*, pages 164–181. Springer-Verlag, 2004.

[2] P. V. Hentenryck et L. Michel : *Constraint-Based Local Search*. MIT Press, 2005.

Chapitre VII

Annexes

VII.1 Model-checking des formules CTL

VII.1.1 Logique CTL

La logique temporelle est utilisée pour exprimer des propriétés du type "chaque requête devra être acquitée", ou bien pour exprimer qu'une assertion est un invariant, c'est à dire qu'elle est toujours vraie. Elle combine la logique combinatoire et des opérateurs temporels. Il existe plusieurs types de logiques temporelles (e.g. LTL ou CTL*), avec des pouvoirs d'expression légèrement différents (voir [7] pour une analyse théorique de différentes logiques temporelles).

La logique "CTL" [6] propose une vision du temps discrète dans laquelle, à chaque instant, le temps peut être découpé en plus d'un futur possible, c'est à dire que l'on considère plusieurs chemins possibles à partir d'un instant donné. La sémantique des formules CTL est définie par rapport à une structure de Kripke $M = (S, S_0, R, L)$ où S est un ensemble d'états, $S_0 \subseteq S$ est l'ensemble des états initiaux, R est une relation binaire totale qui définit les transitions d'états possibles (i.e si le système est dans l'état s à un instant il pourra être à l'instant suivant dans un des états t où $(s,t) \in R$) et L est un étiquetage des états avec des propositions atomiques qui sont vraies sur ces états.

Les formules CTL sont définies récursivement à partir des formules du calcul propositionnel augmenté d'opérateurs temporels. Par exemple, si f_1 et f_2 sont des formules CTL, alors $\mathbf{EX}f_1$ est la formule CTL vraie dans l'état s si et seulement si f_1 est vraie pour un des états successeurs de s, et $\mathbf{E}f_1\mathbf{U}f_2$ est la formule CTL vraie dans l'état s_0 si et seulement si pour un chemin $s_0, s_1, ..., s_i, ..., s_n$, il existe $i \geq 0$ tel que f_1 est vraie dans les états s_0 à s_{i-1} et f_2 est vraie dans l'état s_i. Notons que la logique CTL possède aussi l'opérateur **A** qui indique que la formule doit être vraie pour tout successeur ; comme cet opérateur peut se ramener à l'opérateur **E** en utilisant des négations, je n'ai introduit ici que l'opérateur **E**.

VII.1.2 Model-checking explicite

Etant donné une structure de Kripke $M = (S, S_0, R, L)$ et une formule CTL f, le problème du model-checking de f consiste à montrer qu'il existe un état s pour lequel M est un modèle de f :

$$\exists s \in S,\ M, s \models f$$

La relation $\models$ définit la sémantique formelle des formules CTL. Elle est définie récursivement sur la structure de ces formules. Par exemple, $M, s \models p$ *ssi* $p \in L(s)$, c'est à dire que p est vraie sur l'état s si et seulement si p est un étiquetage de l'état s ; et $M, s \models \mathbf{EX}\ f$ *ssi* $\exists\ t \in S\ tq(s,t) \in R \wedge M, t \models f$, c'est à dire que f est vraie sur un des états successeurs de s.

Quand le nombre d'états dans S est fini, le problème du model-checking de f est décidable. Toutefois, en pratique, le nombre d'états est très grand puisqu'il modélise toutes les configurations possibles des données. Ce problème ne peut donc être résolu en un temps acceptable qu'en utilisant des structures de données efficaces, les BDD et leurs variantes, pour représenter le graphe de transition d'états.

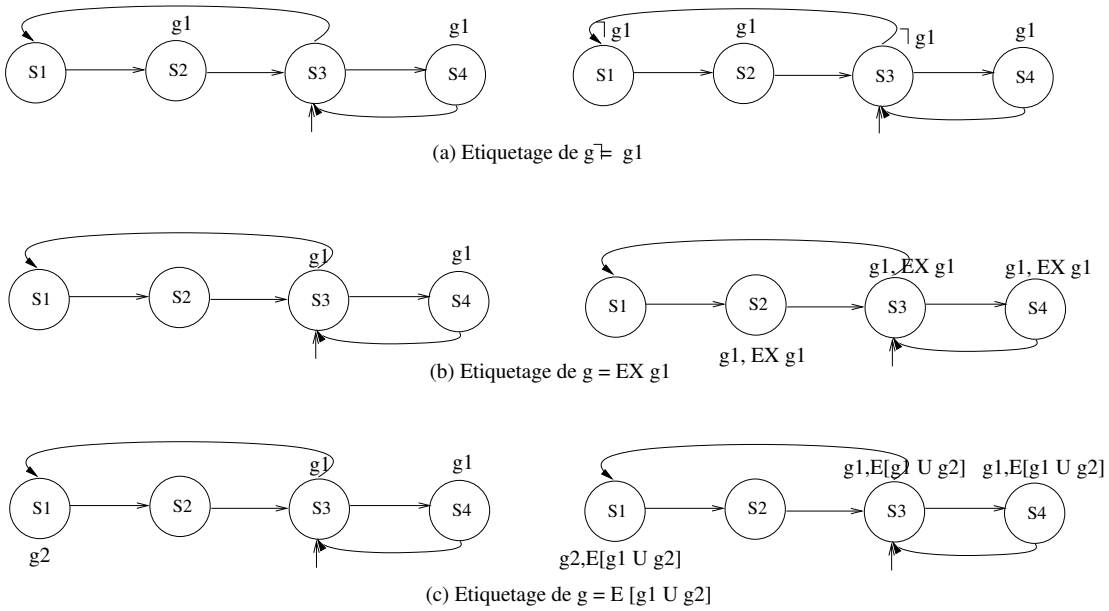


FIG. VII.1 – Exemples d'étiquetage dans l'algorithme de model-checking : **a)** étiquetage de $g = \neg g_1$, **b)** étiquetage de $g = \mathbf{EX}g_1$ et **c)** étiquetage de $g = \mathbf{E}[g_1 \mathbf{U} g_2]$. Les graphes de gauche sont étiquetés avec les sous-formules g_1 et g_2 ; les graphes de droite sont le résultat de l'étiquetage pour g.

L'algorithme le plus simple (i.e model-checking explicite), consiste à effectuer un parcours explicite du graphe de transitions d'état en étiquetant les sommets successivement avec les sous-formules de f, c'est à dire en étiquetant le graphe avec les formules de longueur $i = 1, \ldots, |f|$. L'étiquetage à l'étape $i = 1$ de l'algorithme est donné par l'ensemble L. Pour $i > 1$, soit une sous-formule g de f de longueur i. On peut déterminer si un état doit être étiqueté par g en fonction de la structure de g. Par exemple, si $g = \neg g_1$, tous les états qui ne sont pas étiquetés par g_1 sont étiquetés par g. Pour l'opérateur temporel **EX**, si $g = \mathbf{EX}g_1$, on étiquette un état s avec g si au moins un de ses successeurs est étiqueté par g_1. Pour l'opérateur temporel **U**, l'étiquetage se ramène à un problème d'atteignabilité dans le graphe. Si $g = \mathbf{E}[g_1 \mathbf{U} g_2]$, tous les états étiquetés par g_2 sont étiquetés par g. Puis chaque état qui est étiqueté par g_1 et a un successeur étiqueté par g est aussi étiqueté par g, cette étape étant répétée jusqu'à ce qu'aucun noeud ne puisse être étiqueté par g. La figure VII.1 illustre ces trois exemples d'étiquetages, sur un exemple de graphe tiré du cours de Tevfik Bultan (University of California, [5]).

L'algorithme de model-checking explicite peut nécessiter un parcours du graphe pour chaque sous-formule de f et sa complexité est donc $O(|f| * (|S| + |R|)$.

VII.1.3 BDD : Binary Decision Diagrams

Les diagrammes de décision sont des graphes qui représentent des fonctions dont les variables sont booléennes. Il existe une très grande variété de ces diagrammes [4] : les BDDs, OBDDs, ROBDDs, BMDs, *BMDs,...

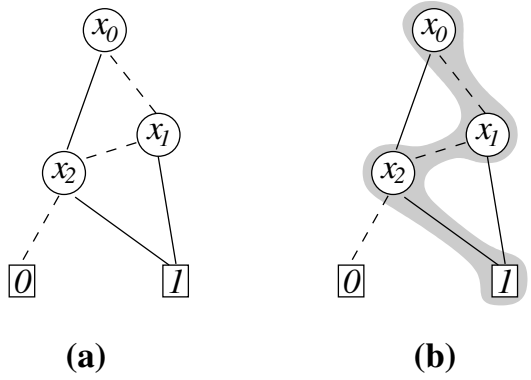


FIG. VII.2 – Le BDD représentant la fonction $(x_0 \wedge x_2) \vee (\neg x_0 \wedge x_1) \vee (\neg x_0 \wedge \neg x_1 \wedge x_2)$.

Ces structures sont particulièrement intéressantes dans les domaines de la CAO [3] comme la vérification formelle, la synthèse, et la génération de vecteurs de test. Elles permettent en particulier de retarder l'explosion combinatoire en espace et sont à la base par exemple du model-checking symbolique ou de l'équivalence observationnelle d'automates.

Nous ne présentons ici que les structures les plus importantes c'est à dire la représentation de base des BDDs et leur optimisation fondamentale les ROBDD. Cette introduction aux BDDs est inspirée du chapitre 12 "Vérification avec les diagrammes de moments binaires" de la thèse de Laurent Arditi, en particulier j'y reprends les mêmes exemples.

Un BDD est un graphe orienté acyclique qui représente une fonction f d'arité n de $\mathbb{B}^n$ dans $\mathbb{B}$. Les nœuds de ce graphe représentent les variables $x_1, \cdots, x_n$. Chaque nœud a deux fils, qui sont eux aussi des BDDs. Les feuilles sont les valeurs 0 et 1. La fonction est évaluée en traversant le graphe à partir de sa racine jusqu'à une feuille. À chaque nœud, un des fils est choisi en fonction de la valeur de la variable racine. Plus précisément, un BDD de racine x_i représente l'expansion de Shannon d'une fonction f par rapport à x_i :

$$f(x_1, \cdots, x_i, \cdots, x_n) = (\neg x_i \wedge f(x_1, \cdots, 0, \cdots, x_n)) \vee (x_i \wedge f(x_1, \cdots, 1, \cdots, x_n))$$

où $f(x_1, \cdots, 0, \cdots, x_n)$ est le *cofacteur négatif* de f par rapport à x_i et $f(x_1, \cdots, 1, \cdots, x_n)$ le *cofacteur positif*.

Le BDD de la figure VII.2 **(a)** représente la fonction $(x_0 \wedge x_2) \vee (\neg x_0 \wedge x_1) \vee (\neg x_0 \wedge \neg x_1 \wedge x_2)$. Chaque arête issue d'un nœud représentant une variable x_i est pleine pour indiquer le fils sélectionné quand $x_i = 1$, et en pointillés pour indiquer le fils quand $x_i = 0$. Si les variables x_0, x_1, x_2 ont les valeurs respectives $0, 0, 1$, alors la valeur de cette fonction est 1 ; il suffit de suivre le chemin indiqué en grisé sur la figure VII.2 **(b)**.

OBDD : Ordered Binary Decision Diagrams Pour être véritablement utilisables, les diagrammes de décision doivent fournir une représentation canonique des

formules booléennes, c'est à dire que deux diagrammes d_1 et d_2 doivent être isomorphes si et seulement si les formules booléennes qu'ils représentent sont égales. Dans [1], Bryant définit les diagrammes de décision binaires ordonnés (OBDDs) comme un sous-ensemble des BDDs. Les OBDDs sont des BDDs dans lesquels les variables sont strictement ordonnées à partir de la racine jusqu'aux terminaux. Il en découle que chaque variable apparaît au plus une fois sur chaque chemin de la racine vers les terminaux. Ainsi, le graphe de la figure VII.2 est un OBDD car on peut prendre l'ordre suivant $x_0 < x_1 < x_2$. L'avantage majeur des OBDDs est la canonicité : pour un ordre donné, il existe une forme canonique pour tout OBDD (que l'on appelle alors ROBDD, R pour "reduced"). Dans la suite, j'emploierai le terme générique de BDDs pour désigner les ROBDDs.

Des algorithmes de manipulation de BDDs permettent l'application de fonctions logiques, la composition (remplacement d'une variable par un BDD), la déterminaison de l'ensemble de satisfaisabilité (ensemble contenant toutes les instanciations telles que la fonction représentée s'évalue à 1). Leur complexité est polynômiale par rapport à la taille des BDDs et ils délivrent des graphes canoniques[1].

De nombreuses implémentations des BDDs ont été réalisées [10], et sont largement utilisées dans les outils de CAO. Mais il reste un inconvénient majeur aux BDDs, c'est que l'ordre des variables influe beaucoup sur leur efficacité. La détermination a priori de l'ordre le plus efficace est un problème NP-difficile [8], et le meilleur ordre peut évoluer au cours de la construction d'un BDD ou de son utilisation. Ainsi certaines implémentations des BDDs réalisent un ordonnancement dynamique des variables [9]. De plus, si la majorité des fonctions sont représentées par des BDDs de façon linéaire par rapport au nombre de variables, il existe des fonctions, notamment la multiplication, dont la taille croît exponentiellement avec le nombre de variables et ce quel que soit l'ordre des variables [2].

VII.2 HOL

VII.2.1 Preuve du théorème d'Euclide

Je présente ici le script ML de la preuve du théorème d'Euclide qui dit que l'ensemble des nombres premiers est infini (i.e `!n. ?p. n < p ∧ prime p`). Cet exemple fait partie de la distribution HOL (répertoire "examples").

```
(*===========================================================================*)
(* Euclid's theorem: for every prime, there is another one that is larger.   *)
(* This proof has been excerpted and adapted from John Harrison's proof of   *)
(* a special case (n=4) of Fermat's Last Theorem.                            *)
(*                                                                           *)
(*===========================================================================*)

(*---------------------------------------------------------------------------*)
(* First, open required context: the theory of arithmetic. This theory is    *)
(* automatically loaded when HOL starts, but the ML module arithmeticTheory  *)
(* needs to be opened before the definitions and theorems of the theory are  *)
(* available without supplying the "arithmeticTheory." prefix.               *)
(*---------------------------------------------------------------------------*)

open arithmeticTheory

(*---------------------------------------------------------------------------*)
(* Divisibility.                                                             *)
(*---------------------------------------------------------------------------*)

val divides_def =
 Define
```

```
  ‘divides a b = ?x. b = a * x‘;

set_fixity "divides" (Infixr 450);

(*---------------------------------------------------------------------------*)
(* Primality.                                                                *)
(*---------------------------------------------------------------------------*)

val prime_def =
 Define
   ‘prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)‘;

(*---------------------------------------------------------------------------*)
(* A sequence of basic theorems about the "divides" relation.                *)
(*---------------------------------------------------------------------------*)

val DIVIDES_0 = store_thm
 ("DIVIDES_0",
  ‘‘!x. x divides 0‘‘,
  METIS_TAC [divides_def,MULT_CLAUSES]);

val DIVIDES_ZERO = store_thm
 ("DIVIDES_ZERO",
  ‘‘!x. 0 divides x = (x = 0)‘‘,
  METIS_TAC [divides_def,MULT_CLAUSES]);

val DIVIDES_ONE = store_thm
 ("DIVIDES_ONE",
  ‘‘!x. x divides 1 = (x = 1)‘‘,
  METIS_TAC [divides_def,MULT_CLAUSES,MULT_EQ_1]);

val DIVIDES_REFL = store_thm
 ("DIVIDES_REFL",
  ‘‘!x. x divides x‘‘,
  METIS_TAC [divides_def,MULT_CLAUSES]);

val DIVIDES_TRANS = store_thm
 ("DIVIDES_TRANS",
  ‘‘!a b c. a divides b /\ b divides c ==> a divides c‘‘,
  METIS_TAC [divides_def,MULT_ASSOC]);

val DIVIDES_ADD = store_thm
("DIVIDES_ADD",
 ‘‘!d a b. d divides a /\ d divides b ==> d divides (a + b)‘‘,
 METIS_TAC[divides_def,LEFT_ADD_DISTRIB]);

val DIVIDES_SUB = store_thm
 ("DIVIDES_SUB",
  ‘‘!d a b. d divides a /\ d divides b ==> d divides (a - b)‘‘,
  METIS_TAC [divides_def,LEFT_SUB_DISTRIB]);

val DIVIDES_ADDL = store_thm
 ("DIVIDES_ADDL",
  ‘‘!d a b. d divides a /\ d divides (a + b) ==> d divides b‘‘,
  METIS_TAC [ADD_SUB,ADD_SYM,DIVIDES_SUB]);

val DIVIDES_LMUL = store_thm
 ("DIVIDES_LMUL",
  ‘‘!d a x. d divides a ==> d divides (x * a)‘‘,
  METIS_TAC [divides_def,MULT_ASSOC,MULT_SYM]);

val DIVIDES_RMUL = store_thm
 ("DIVIDES_RMUL",
  ‘‘!d a x. d divides a ==> d divides (a * x)‘‘,
  METIS_TAC [MULT_SYM,DIVIDES_LMUL]);

val DIVIDES_LE = store_thm
 ("DIVIDES_LE",
  ‘‘!m n. m divides n ==> m <= n \/ (n = 0)‘‘,
  RW_TAC arith_ss  [divides_def]
     THEN Cases_on ‘x‘
     THEN RW_TAC arith_ss  [MULT_CLAUSES]);

(*---------------------------------------------------------------------------*)
(* Various proofs of the same formula                                        *)
(*---------------------------------------------------------------------------*)

val DIVIDES_FACT = store_thm
 ("DIVIDES_FACT",
  ‘‘!m n. 0 < m /\ m <= n ==> m divides (FACT n)‘‘,
  RW_TAC arith_ss  [LESS_EQ_EXISTS]
   THEN Induct_on ‘p‘
   THEN RW_TAC arith_ss  [FACT,ADD_CLAUSES]
   THENL [Cases_on ‘m‘, ALL_TAC]
   THEN METIS_TAC [FACT, DECIDE ‘‘!x. ~(x < x)‘‘,
                   DIVIDES_RMUL, DIVIDES_LMUL, DIVIDES_REFL]);

val DIVIDES_FACT = prove
```

```
 (‘‘!m n. 0 < m /\ m <= n ==> m divides (FACT n)‘‘,
  RW_TAC arith_ss  [LESS_EQ_EXISTS]
   THEN Induct_on ‘p‘
   THEN RW_TAC arith_ss  [FACT,ADD_CLAUSES]
   THEN METIS_TAC [FACT, DECIDE ‘‘!x. ~(x < x)‘‘, num_CASES,
                   DIVIDES_RMUL, DIVIDES_LMUL, DIVIDES_REFL]);

val DIVIDES_FACT = prove
 (‘‘!m n. 0 < m /\ m <= n ==> m divides (FACT n)‘‘,
  RW_TAC arith_ss  [LESS_EQ_EXISTS]
   THEN Induct_on ‘p‘
   THEN METIS_TAC [FACT, DECIDE ‘‘!x. ~(x < x)‘‘, num_CASES,
                   DIVIDES_RMUL,DIVIDES_LMUL,DIVIDES_REFL,ADD_CLAUSES]);


val DIVIDES_FACT = prove
(‘‘!m n. 0 < m /\ m <= n ==> m divides (FACT n)‘‘,
Induct_on ‘n - m‘
 THEN RW_TAC arith_ss  [] THENL
 [‘m = n‘          by DECIDE_TAC THEN
  ‘?k. m = SUC k‘ by METIS_TAC[num_CASES,prim_recTheory.LESS_REFL]
     THEN METIS_TAC[FACT,DIVIDES_RMUL,DIVIDES_REFL],
  ‘0 < n‘          by DECIDE_TAC THEN
  ‘?k. n = SUC k‘ by METIS_TAC [num_CASES,prim_recTheory.LESS_REFL]
   THEN RW_TAC arith_ss  [FACT, DIVIDES_RMUL]]);


(*---------------------------------------------------------------------------*)
(* Zero and one are not prime, but two is.  All primes are positive.         *)
(*---------------------------------------------------------------------------*)

val NOT_PRIME_0 = store_thm
 ("NOT_PRIME_0",
  ‘‘~prime 0‘‘,
  RW_TAC arith_ss  [prime_def,DIVIDES_0]);

val NOT_PRIME_1 = store_thm
 ("NOT_PRIME_1",
  ‘‘~prime 1‘‘,
  RW_TAC arith_ss  [prime_def]);

val PRIME_2 = store_thm
 ("PRIME_2",
  ‘‘prime 2‘‘,
  RW_TAC arith_ss  [prime_def] THEN
  METIS_TAC [DIVIDES_LE, DIVIDES_ZERO,
             DECIDE‘‘~(2=1) /\ ~(2=0) /\ (x<=2 = (x=0) \/ (x=1) \/ (x=2))‘‘]);

val PRIME_POS = store_thm
 ("PRIME_POS",
  ‘‘!p. prime p ==> 0<p‘‘,
  Cases THEN RW_TAC arith_ss [NOT_PRIME_0]);

(*---------------------------------------------------------------------------*)
(* Every number has a prime factor, except for 1. The proof proceeds by a    *)
(* "complete" induction on n, and then considers cases on whether n is       *)
(* prime or not. The first case (n is prime) is trivial. In the second case, *)
(* there must be an "x" that divides n, and x is not 1 or n. By DIVIDES_LE,  *)
(* n=0 or x <= n. If n=0, then 2 is a prime that divides 0. On the other     *)
(* hand, if x <= n, there are two cases: if x<n then we can use the i.h. and *)
(* by transitivity of divides we are done; otherwise, if x=n, then we have   *)
(* a contradiction with the fact that x is not 1 or n.                       *)
(*---------------------------------------------------------------------------*)

val PRIME_FACTOR = store_thm
 ("PRIME_FACTOR",
  ‘‘!n. ~(n = 1) ==> ?p. prime p /\ p divides n‘‘,
  completeInduct_on ‘n‘
   THEN RW_TAC arith_ss  []
   THEN Cases_on ‘prime n‘ THENL
   [METIS_TAC [DIVIDES_REFL],
    ‘?x. x divides n /\ ~(x=1) /\ ~(x=n)‘ by METIS_TAC[prime_def] THEN
    METIS_TAC [LESS_OR_EQ, PRIME_2,
               DIVIDES_LE, DIVIDES_TRANS, DIVIDES_0]]);

(*---------------------------------------------------------------------------*)
(* In the following proof, METIS_TAC automatically considers cases on        *)
(* whether n is prime or not.                                                *)
(*---------------------------------------------------------------------------*)

val PRIME_FACTOR = prove
 (‘‘!n. ~(n = 1) ==> ?p. prime p /\ p divides n‘‘,
  completeInduct_on ‘n‘ THEN
  METIS_TAC [DIVIDES_REFL,prime_def,LESS_OR_EQ, PRIME_2,
             DIVIDES_LE, DIVIDES_TRANS, DIVIDES_0]);


(*---------------------------------------------------------------------------*)
```

```
(* Every number has a prime greater than it.                                *)
(* Proof.                                                                   *)
(* Suppose not; then there's an n such that all p greater than n are not    *)
(* prime. Consider FACT(n) + 1: it's not equal to 1, so there's a prime q   *)
(* that divides it. q also divides FACT n because q is less-than-or-equal   *)
(* to n. By DIVIDES_ADDL, this means that q=1. But then q is not prime,     *)
(* which is a contradiction.                                                *)
(*--------------------------------------------------------------------------*)

val EUCLID = store_thm
 ("EUCLID",
  ``!n. ?p. n < p /\ prime p``,
  SPOSE_NOT_THEN STRIP_ASSUME_TAC
   THEN MP_TAC (SPEC ``FACT n + 1`` PRIME_FACTOR)
   THEN RW_TAC arith_ss  [FACT_LESS, DECIDE ``~(x=0) = 0<x``]
   THEN METIS_TAC [DIVIDES_FACT, DIVIDES_ADDL, DIVIDES_ONE,
                   NOT_PRIME_1, NOT_LESS, PRIME_POS]);


(*--------------------------------------------------------------------------*)
(* The previous proof is somewhat unsatisfactory, because its structure gets *)
(* hidden in the invocations of the automated reasoners. An assertional     *)
(* style allows a presentation that mirrors the informal proof.             *)
(*--------------------------------------------------------------------------*)

val EUCLID_AGAIN = prove (``!n. ?p. n < p /\ prime p``,
CCONTR_TAC
THEN
   `?n. !p. n < p ==> ~prime p`  by METIS_TAC[]              THEN
   `~(FACT n + 1 = 1)`           by RW_TAC arith_ss [FACT_LESS,
                                    DECIDE ``~(x=0) = 0<x``] THEN
   `?p. prime p /\
        p divides (FACT n + 1)`  by METIS_TAC [PRIME_FACTOR] THEN
   `0 < p`                       by METIS_TAC [PRIME_POS]    THEN
   `p <= n`                      by METIS_TAC [NOT_LESS]     THEN
   `p divides FACT n`            by METIS_TAC [DIVIDES_FACT] THEN
   `p divides 1`                 by METIS_TAC [DIVIDES_ADDL] THEN
   `p = 1`                       by METIS_TAC [DIVIDES_ONE]  THEN
   `~prime p`                    by METIS_TAC [NOT_PRIME_1]
THEN
METIS_TAC[]);
```

Bibliographie

[1] Randal E. Bryant : Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, août 1986.

[2] Randal E. Bryant : On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, février 1991.

[3] Randal E. Bryant : Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), septembre 1992.

[4] Randal E. Bryant : Binary decision diagrams and beyond : Enabling technologies for formal verification. *In IEEE Internationl Conference on Computer-Aided Design*, pages 236–243, San Jose, CA (USA), 1995. IEEE Computer Society Press.

[5] Tevfik Bultan : Cours cs 267 - automated verification, university of california. `http://www.cs.ucsb.edu/~bultan/courses/267/`.

[6] Emerson E. Clarke E. et Sistla A. : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8,2:224–263, 1986.

[7] Emerson E. : Temporal and modal logic. *Handbook of Theoretical Computer Science*, B:995–1072, 1990.

[8] S.J. Friedman et K.J. Supowit : Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, C(39):710–713, 1990.

[9] Richard Rudell : Dynamic variable ordering for ordered binary decision diagrams. *In IEEE Internationl Conference on Computer-Aided Design*, pages 24–47, Santa Clara, CA (USA), novembre 1993. IEEE Computer Society Press.

[10] Ellen M. Sentovitch : A study of the performance of BDD packages. *In* Srivas et Camilleri, éditeurs : *International Conference on Formal Methods in Computer-Aided Design*, volume 1166 de *Lecture Notes in Computer Science*, Palo Alto, CA (USA), 1996.

Deuxième partie

Curriculum Vitae

État civil

Hélène Collavizza

Née le 3 juin 1964 à Montfavet (84)

Adresse personnelle

Les jardins de Mouans III
31 traverse des écureuils, Bat F
06370 Mouans Sartoux

Adresse professionnelle

Polytech'Nice Sophia-Antipolis, département informatique
930 Route des Colles - BP 145
06903 Sophia-Antipolis Cedex
04 92 96 51 63
helen@polytech.unice.fr

Titres universitaires

Thèse de doctorat, Université de Provence, Aix-Marseille I

Janvier 1991
Spécialité : Mathématiques Appliquées
Option : Informatique
Titre : « Sémantique Fonctionnelle des Microprocesseurs :
l'Environnement de Spécification et de Preuve μSPEED »
Directrice : Dominique Borrione, Université Joseph Fourier, Grenoble
Rapporteurs : Paul Caspi, CNRS INPG-LGI Grenoble
Daniel Le Métayer, IRISA Rennes
Pierre Lescanne, CRIN Nancy

DEA de Mathématiques Appliquées Université de Provence, Aix-Marseille I, 1986-1987 (mention Bien)

Fonction actuelle Maître de conférences à l'Ecole Polytechnique Universitaire de Nice Sophia-Antipolis (ex ESSI) depuis 1992.

Chapitre I

Activités d'enseignement

1 Présentation générale

1.1 Enseignements à Polytech'Nice Sophia-Antipolis

Mes enseignements se déroulent principalement dans mon UFR de rattachement, le département informatique de Polytech'Nice Sophia-Antipolis en $1^{ère}$ et $2^{ème}$ année du cycle ingénieur (notés respectivement SI'3 et SI'4 pour "Science Informatique" niveau 3 et 4 car les niveaux 1 et 2 correspondent aux deux années de cycle préparatoire). Les points forts de mon activité à Polytech'Nice Sophia-Antipolis ont été la responsabilité du cours d'architecture des ordinateurs, que j'ai introduit peu après mon arrivée, et la mise en place avec mon collègue Jean-Paul Stromboni de projets associés aux journées DeViNT, où les élèves créent des logiciels dédiées aux enfants déficients visuels (voir 1.2). J'ai également donné des cours intégrés de logique et des TD machine en programmation réseau, et des cours intégrés d'algorithmique. Depuis mon retour de délégation CNRS, j'effectue des TD en programmation Java pour des publics nouveaux pour moi : les élèves de deuxième année du cycle préparatoire (CIP'2) et les élèves en $1^{ère}$ année de cycle ingénieur du département mathématiques appliquées (MAM'3). Mon implication dans les activités d'enseignement m'a amenée régulièrement à dépasser mes obligations statutaires.

1.2 Projets des journées DeViNT

La journée DeViNT (voir `http://devint.polytech.unice.fr/`) a été initiée en 2003 lors de l'année des handicapés par ma collègue Mireille Blay. L'objectif est de rassembler déficients visuels, associations, élus, enseignants et chercheurs afin d'analyser les problèmes rencontrés par les déficients visuels et de montrer comment les nouvelles technologies de l'information et de la communication peuvent aider à les résoudre. L'audience de la $7^{ème}$ journée DeViNT en 2009, dont le thème était "Éducation, Formation et Nouvelles Technologies", a été d'une centaine de participants, dont un tiers de déficients visuels.

Depuis 2004, en collaboration avec Jean-Paul Stromboni, je coordonne des projets de première année liés à la journée DeViNT (voir `http://projets.polytech.`

`unice.fr/Devint/`). Ces projets sont une expérience pédagogique innovante, où nos étudiants sont confrontés à de vrais utilisateurs, qui ont de plus une spécificité forte puisqu'ils sont déficients visuels. L'objectif est de développer des logiciels éducatifs ou ludiques qui soient adaptés à des enfants ou adolescents déficients visuels. Depuis 2004, nous avons instauré une collaboration étroite avec l'Institut d'Education Sensorielle C. Ader et l'école spécialisée du château à Nice. Les éducateurs nous proposent des thèmes, nous donnent des consignes à suivre pour les interfaces. Nos étudiants font un premier prototype puis ils se rendent à l'IES C. Ader pour faire tester leurs logiciels par les enfants et les éducateurs. Cette expérience forme nos étudiants à savoir écouter, comprendre et devancer les besoins de l'utilisateur. Ils font preuve pour ces projets d'une motivation et d'un sérieux exemplaire.

Les projets DeViNT sont pour moi une activité pédagogique passionnante mais qui nécessite un fort investissement. En effet, outre le suivi de projet usuel, nous devons gérer une organisation logistique importante : lien entre les étudiants et les éducateurs de l'école du château et de l'IES C. Ader, achat de matériel spécifique, gestion de la visite à l'IES C. Ader (30 étudiants environ en déplacement), gestion des stands lors de la journée DeViNT, gestion de la diffusion des jeux auprès des déficients visuels.

J'ai d'autre part encadré de nombreux projets autour de la déficience visuelle en SI'4 (voir section 2.3). Les étudiants développent des boîtes à outil spécifiques qui sont ensuite utilisées pour créer et développer les projets de$1^{ère}$ année. Ces projets sont devenus de véritables «projets d'école», utilisés et améliorés par plusieurs promotions.

Le chapitre III présente les activités de valorisation et de diffusion liées aux projets DeViNT, qui ont fait l'objet de plusieurs présentations à des colloques.

1.3 Enseignements dans d'autres UFR

J'ai été responsable de septembre 2001 à septembre 2007, date de ma délégation CNRS, du cours d'algorithmique dans la licence professionnelle LPSIL, à l'IUT d'informatique (volume annuel de l'ordre de 40h TD). L'approche pédagogique que j'ai choisie est très différente de celle des cours d'algorithmique en $1^{ère}$ année de Polytech'Nice Sophia-Antipolis. Les étudiants ayant pour la plupart une bonne pratique de la programmation, l'accent est donc mis sur la prise de recul par rapport aux programmes écrits afin de caractériser leur correction et leur efficacité. Les notions de mathématiques discrètes nécessaires à l'évaluation des complexités sont introduites au fur et à mesure qu'elles deviennent nécessaires.

D'autre part, j'ai participé deux années consécutives à des enseignements à l'UFR LASH (Langues Art Sciences Humaines) en algorithmique pour le DEUST informatique et en introduction à l'algorithmique en java-script pour la licence professionnelle IM@SH (Informatique Multimédia et Sciences Humaines).

2 Détail des enseignements

Je donne ici le détail de mes enseignements par ordre chronologique inverse. Les volumes horaires sont donnés en équivalent TD à titre indicatif (valeur moyenne

annuelle).

2.1 Service actuel

Depuis mon retour de délégation CNRS, j'ai choisi de changer de public en effectuant des TD aux MAM'3 et en CIP'2. Je continue à encadrer les projets DeViNT avec mon collègue Jean-Paul Stromboni et je continue les TD d'algorithmique en SI'3 (enseignement que j'ai toujours plaisir à faire, car il s'agit d'une matière qui est délicate à faire comprendre, et il faut donc adapter son discours chaque année). Mon service est donc composé comme suit :

Algorithmique et programmation en Java en CIP'2 (cours intégrés TD/TP, 86hTD en 2009-2010).

Algorithmique et programmation en Java en MAM'3 (TD/TP, rédaction de sujets spécifiques à la filière MAM, 52hTD/an depuis février 2009).

Projets des journées DeViNT, en collaboration avec Jean-Paul Stromboni (48hTD/an).

Algorithmique en SI'3 (TD, 52hTD/an).

2.2 Cours/TD/TP effectués depuis ma nomination

Algorithmique en SI'3 (TD, 70hTD/an, depuis 1992, cours intégrés en 2006-2007)

Architecture des ordinateurs / assembleur en SI'3 (responsable du cours, cours/TD/TP 80hTD/an, de 1995 à 2007)

Logique en SI'4 (TD de 1992 à 2007, responsable du cours depuis 1999, 30 hTD/an)

Internet et réseau en java en SI'4 (TP de 2004 à 2007, 24h/an)

Algorithmique en Licence Professionnelle des Métiers Informatiques, (responsable du cours de 2001 à 2007, cours/TD 40h/an)

Algorithmique et java-script en licence professionnelle IM@SH (en 2005, 20h de cours intégrés)

Programmation en JAVA en SI'3 (TP de 1997 à 1999, 56h TD/an)

Mathématiques discrètes en SI'3 (TD de 1996 à 1998, 24hTD/an)

Module "Langage et Architectures Parallèles" en SI'5 (cours de 1992 à 1995)

Cours intégrés en harmonisation d'architecture des ordinateurs en SI'3 (30h/an, responsable du cours, de 1993 à 1996)

Cours dans le module "Démonstration Automatique" du DEA informatique (de 1993 à 1996)

2.3 Encadrement de projets

Projets sur le semestre en SI'3 *Il s'agit de projets répartis sur le semestre, à raison de deux heures par semaine, pour toute la promotion soit entre 70 et 100 élèves.*

Depuis 2004, en collaboration avec Jean-Paul Stromboni, projets des journées DeViNT (voir 1.2, 48hTD/an).

En 2003, projet d'algorithmique paradigme "Diviser pour régner" : les gratte-ciel.

En 2002, simulation d'architectures de type chemin de données et partie contrôle en Java.

Projets bloqués en SI'3 *Il s'agit de projets d'une semaine à temps plein pour toute la promotion.*

De 2002 à 2007, en collaboration avec Marc Gaëtano, projets algorithmique et Java.

En 2000 : applet Java de simulation des processus adaptatifs étudiés dans le module d'automatique, en collaboration avec Jean-Paul Stromboni.

De 1992 à 1995 : programmation en assembleur de routines système (responsable du projet).

Projets SI'4 *Il s'agit de projets de 3 semaines à temps complet effectués par des groupes de 3 à 4 étudiants.* Les projets qui ont un lien avec les projets DeViNT sont précédés de [DeViNT].

En 2007, plug-in eclipse de traduction de Java vers XML en utilisant l'API JDT.

[DeViNT] En 2007, en collaboration avec Jean-Paul Stromboni, site web de diffusion des projets DeViNT.

[DeViNT] En 2007, en collaboration avec Jean-Paul Stromboni, faire chanter la synthèse vocale.

[DeViNT] En 2007, en collaboration avec Jean-Paul Stromboni, développement d'une interface pour un laboratoire de prosodie pour la synthèse vocale SI_VOX.

[DeViNT] En 2007, en collaboration avec Jean-Paul Stromboni, amélioration de l'algorithme de génération de la prosodie pour la synthèse vocale SI_VOX.

[DeViNT] En 2006, en collaboration avec Jean-Paul Stromboni et Stéphane Lavirotte, service web pour accéder via une API commune à la synthèse vocale réalisée à Polytech'Nice Sophia-Antipolis et à la Synthèse vocale commerciale Acapela.

[DeViNT] En 2006, en collaboration avec Jean-Paul Stromboni, diffusion des projets DeViNT : définition d'un outil de création de CDRom contenant les projets sélectionnés de l'année.

En 2006, en collaboration avec Michel Rueher et Claude Michel, utilisation des contraintes pour des programmes corrects dans l'environnement Java.

En 2004, traitement d'un fichier GEDCOM pour le stockage d'arbres généalogiques.

[DeViNT] En 2004, en collaboration avec Anne-Marie Pinna, intégration de signets électroniques pour la lecture pour les déficients visuels.

[DeViNT] En 2004, en collaboration avec Jean-Paul Stromboni et Gérard Uzan du laboratoire d'ergonomie de Paris VI, organisation hiérarchique des liens dans les pages web pour l'accès des utilisateurs aveugles.

[DeViNT] En 2004, en collaboration avec Jean-Paul Stromboni et Stéphane Lavirotte, installateur des projets DeViNT accessible aux déficients visuels.

[DeViNT] En 2003, une synthèse vocale destinée aux applications pour les déficients visuels. Cette synthèse est utilisée par la plupart des projets SI'3 pour la journée DeViNT. Les principes de mise en oeuvre et ses applications sont présentés section III.

En 2001, portail WEB pour une base de données de postures et séances de yoga.

En 1993, projet de maîtrise : simulateur C++ pour l'architecture des ordinateurs

Projet du module IHM (SI'5) En 2005, en collaboration avec Anne-Marie Pinna, introduction du son dans les interfaces dédiées aux déficients visuels.

Chapitre II

Responsabilités administratives

1 Responsabilités administratives les plus importantes

Depuis ma nomination comme maître de conférences, j'ai toujours assuré des responsabilités administratives importantes. Je mentionne ici celles qui m'ont demandé le plus de disponibilité.

1.1 Responsable pédagogique de la $1^{ère}$ année d'école d'ingénieur

De janvier 2000 à janvier 2004 j'ai été responsable pédagogique d'ESSI $1^{ère}$ année, pour des promotions variant entre 100 et 120 étudiants. Ma tâche consistait à organiser la pédagogie des enseignements et le suivi des étudiants, à gérer les emplois du temps, à préparer et diriger les jurys. Cette responsabilité administrative a été d'autant plus lourde que j'ai dû assurer le transfert de compétences du secrétariat pédagogique, suite au départ de la secrétaire qui en assurait la totalité depuis près de dix ans. D'autre part, l'ESSI était en cours de réorganisation pour devenir le département informatique de Polytech'Nice Sophia-Antipolis, ce qui a nécessité de nombreuses réunions du comité pédagogique. Durant ces quatre années de responsabilité j'ai porté mes efforts sur plusieurs axes prioritaires :

- Mise en place d'une nouvelle grille des programmes (réduction des heures de cours en amphithéâtre au profit de cours intégrés, harmonisation de la répartition thématique entre mathématiques, informatique, électronique, sciences humaines),
- Renforcement des créneaux de projets, les étudiants ayant de plus en plus besoin d'être confrontés à des applications concrètes pour pouvoir mettre en oeuvre leurs acquis théoriques,
- Renforcement des matières associées selon la directive de la Commission du Titre d'Ingénieur (augmentation des heures d'anglais et de technique de communication, mise en place d'un cours de chinois en 2000),
- Soutien aux étudiants en difficulté : créneaux de soutien, mise en place d'un système de tutorat, mise en place de bilans de compétences et d'une aide à la réorientation pour les étudiants exclus de l'école,
- Mise en place des contenus du serveur WEB de description des cours.

1.2 Membre du bureau de la CS $27^{ème}$ section

J'ai participé à trois commissions de spécialiste de $27^{ème}$ section, en tant qu'assesseur de janvier 1994 à avril 1998, comme vice-présidente maître de conférences d'avril 1998 à Octobre 2001 puis comme membre simple de Septembre 2004 à Octobre 2008. La responsabilité de vice-présidente maître de conférences a exigé un grand investissement : dépouillement des dossiers et rédaction de fiches candidat, participation à toutes les commissions d'audition, organisation logistique des commissions et enfin réalisation d'un serveur Web pour le dépôt des rapports.

1.3 Coprésidente de la journée DeViNT 2006 (avec Stéphane Lavirotte et Jean l'Herbon de Lussats)

En 2006, j'ai été coprésidente de la journée DeViNT'2006 dont le thème était «Art, culture et loisirs». Comme toute conférence, organiser DeViNT nécessite de définir le programme, de demander des subventions, et de gérer l'organisation. Mais les tâches habituelles de l'organisation logistique d'une conférence sont rendues plus difficiles du fait du public handicapé. La gestion de la logistique (que j'ai assurée également en 2005 et 2004) a été définie en collaboration avec des aveugles qui ont mis en évidence les pièges pratiques à éviter (comme des pas de portes en relief et par conséquent dangereux). A l'initiative d'Anne-Marie Pinna, nous avons mis en place et géré une équipe d'une vingtaine d'étudiants pour accompagner les déficients visuels tout au long de la journée.

1.4 Responsable de la communication de l'école

De Septembre 1995 à Janvier 2000 j'ai été responsable de la communication du département informatique. Cette tâche m'a pris beaucoup de temps. En particulier, nous avons redéfini de nouveaux supports de communication : plaquettes, affiches, logo (avec Jean-Paul Rigault qui en était le directeur). Il m'a fallu également négocier avec les annonceurs (e.g. le journal "L'étudiant"), rédiger les présentations, et organiser la présentation de l'école aux différents forums étudiants et journées d'orientation en classes préparatoires.

2 Liste des responsabilités administratives

- Membre élu du conseil du département informatique de Polytech'Nice Sophia-Antipolis (de juin 2005 à septembre 2007, réunions bi-mensuelles). Animatrice du groupe de réflexion sur la pédagogie pour le plan quadriénal.
- Correspondant de la Bibliothèque Universitaire (Septembre 2004 à septembre 2007). Achat des livres, gestion de la mise en ligne des livres du service Safari, participation aux commissions documentaires, aide à la gestion du personnel de l'antenne de Sophia-Antipolis.
- Coprésidente de la journée DeViNT'2006 (avec S. Lavirotte et Jean l'Herbon de Lussats) : définition du programme, demande de subventions, gestion des orateurs, gestion de la communication, gestion des participants et du budget, définition d'une charte, participation à des réunions sur le handicap.

- Responsable de la logistique pour les journées DeViNT 2005 et 2004 : gestion des aspects matériels (repas, achat de matériel, signalisation, organisation des transports, ...) et gestion des étudiants volontaires pour aider à l'organisation de la journée.
- Membre de la Commission de Spécialistes $27^{ème}$ section : assesseur de septembre 1994 à avril 1998, vice-présidente maître de conférences d'avril 1998 à Octobre 2001, membre simple de Septembre 2004 à Octobre 2008.
- Responsable pédagogique de la première année de l'ESSI (de Janvier 2000 à janvier 2004) : promotions variant entre 100 et 120 étudiants, gestion des emplois du temps, organisation pédagogique des enseignements (mise en place d'une nouvelle grille en 2001), organisation des jurys et suivi des étudiants en échec, interlocutrice privilégiée auprès des étudiants, gestion des stages ouvriers, représentant de l'ESSI auprès de la LPMI à l'UNSA.
- Membre élu du Conseil d'Administration de l'ESSI (de 1994 à 2000)
- Responsable de la communication de l'Ecole (de Septembre 1995 à Janvier 2000)
- Responsable des séminaires ESSI-3 (de 1994 à 1996)

Chapitre III

Activités de valorisation et de diffusion

Je présente dans ce chapitre mes activités de valorisation et de diffusion qui sont à mis chemin entre enseignement et recherche. En effet, ces activités ont émergé des projets DeViNT (voir section 1.2 page 281) et ont également donné lieu à une publication et à des présentations dans des colloques. Ce travail a été réalisé en collaboration étroite et enthousiaste avec mon collègue Jean-Paul Stromboni.

1 Implémentation et diffusion d'une synthèse vocale

Pour mettre en œuvre les interfaces des logiciels dédiés aux déficients visuels, il s'est très vite avéré indispensable de compenser les informations visuelles par des informations sonores. Nous avons donc recherché un logiciel de synthèse vocale qui soit diffusable gratuitement et facilement utilisable par nos étudiants. Les synthèses vocales existantes étant pour la plupart commerciales, ou mal adaptées à notre problème, nous avons décidé d'en développer une. Après avoir défini un premier prototype via un projet d'étudiants de quatrième année, j'ai consolidé une version qui est largement diffusée sur le site des projets de l'école [1]. Les principes de cette synthèse sont les suivants :

- elle s'appuie sur le *synthétiseur MBROLA* développé à l'université de Mons (voir `http://tcts.fpms.ac.be/synthesis/mbrola.html`). Ce synthétiseur prend en entrée un fichier de description de phonèmes qui décrit à la fois le son à prononcer (i.e le phonème), sa fréquence (pour produire un son plus ou moins aigu) ainsi que sa durée. Le synthétiseur génère en sortie un fichier au format wave.
- un *module de phonétisation* prend en entrée un texte et le traduit en une suite de phonèmes. Il utilise un ensemble de règles de prononciation données dans des fichiers, et stockées sous forme d'expressions régulières. Afin d'être efficace, ces fichiers sont chargés en mémoire sous la forme d'un arbre lexical, qui est parcouru récursivement pendant la traduction du texte à phonétiser.

[1] La synthèse peut être télé-chargée à http ://vocalyse.polytech.unice.fr/. Il y a environ un télé-chargement par semaine sur ce serveur depuis janvier 2007

Les fichiers de règles peuvent être mis à jour de façon simple par l'utilisateur, afin par exemple d'ajouter une nouvelle abréviation, un nom propre, ou un nouveau mot dont les règles de prononciation génériques n'assurent pas une prononciation correcte.
- un *module de prosodie* associe une "musique" au texte. Ce module associe une courbe d'intonation en fonction de différents schémas de phrases (interrogative, exclamative, ...). Pour cela, les phrases sont découpées en syntagmes et l'intonation est calculée en fonction du type et de la longueur du syntagme.

Les modules de phonétisation et de prosodie ont été entièrement développés à l'école. Le module de prosodie a nécessité un effort particulier pour obtenir une intonation satisfaisante ; il a été amélioré à plusieurs reprises lors de projets étudiants. La version actuelle est largement acceptable (au dire des enfants). Elle a été conçue en analysant les courbes du signal associées à un corpus de phrases lues par une synthèse vocale commerciale, afin de déterminer des règles d'intonation.

J'ai présenté les principes de cette synthèse vocale aux XXVIème « Journées d'Etudes sur la Parole » en juin 2006. La synthèse est utilisée dans les projets DeViNT et dans de nombreuses applications, comme par exemple un environnement de simulateur de vol, ou le logiciel «DonnerLaParole» diffusé sur `http://donnerlaparole.sourceforge.net`.

2 Diffusion de logiciels pour les enfants déficients visuels

Grâce à notre collaboration étroite avec l'Institut d'Education Sensorielle C. Ader et l'école spécialisée du château à Nice dans le cadre des projets DeViNT, nous avons acquis de plus en plus d'expertise concernant la conception d'interfaces dédiées aux déficients visuels. Nous avons pu ainsi recueillir de nombreuses consignes pour rendre nos jeux plus accessibles comme par exemple :

- choisir des raccourcis clavier usuels des logiciels de lecture d'écran (comme par exemple le logiciel "Jaws Job Access With Speech" très largement utilisé par les déficients visuels),
- associer une information sonore à toute information visuelle, sans pour autant trop ralentir le rythme du jeu,
- utiliser des dessins aux bordures simples et contrastées,
- ne mettre une information visuelle que si elle est indispensable (pas de décors inutiles),
- assurer que les objets (e.g. une balle) se déplacent suffisamment lentement. Il faut par exemple pouvoir suivre l'objet en collant son visage près de l'écran, ce qui est nécessaire pour certaines déficiences où le champ de vision est très restreint,
- offrir la possibilité de configurer l'interface du jeu ; pour cela, quelques profils qui tiennent compte des principaux handicaps suffisent.

Afin d'inciter les étudiants à suivre ces consignes d'interface, j'ai écrit une bibliothèque Java pour développer les jeux. Un ensemble de classes Java, associée à une structure bien précise pour déposer les ressources du jeu (images, sons) ou bibliothèques spécifiques ont été définies. Les étudiants développent leurs projets à partir de ces briques de base. Cela a deux intérêts majeurs :

- Les interfaces de l'ensemble des projets sont uniformes. Ainsi quand les jeux sont diffusés sur CD-ROM, l'utilisateur n'est pas perturbé par des différences de conventions entre les jeux.
- La structure logicielle imposée a facilité le développement d'un outil de génération automatique d'une image des jeux DeViNT qui peut ensuite être gravée sur CD-ROM.

Depuis 2006, un CD-ROM contenant la plupart des projets a été diffusé auprès des participants déficients visuels lors de la journée DeViNT (cette journée compte une centaine de participants dont plus d'un tiers sont déficients visuels). A notre grande surprise et je dois l'avouer fierté, nous avons constaté lors de notre visite à l'institut Clément Ader cette année que les élèves connaissent les jeux DeViNT "par cœur" : il suffit d'évoquer le nom d'un jeu pour qu'ils puissent dire à quoi il correspond. En particulier, nous avons évoqué de supprimer un jeu pour l'édition 2008-2009 et un des enfants s'est insurgé car il s'agissait de son jeu préféré.

3 Diffusion dans des colloques

Je liste ici les colloques dans lesquels la synthèse vocale et les projets DeViNT ont été présentés.

- La synthèse vocale a été présentée aux XXVIème «Journées d'Etudes sur la Parole» en juin 2006. La publication est la suivante :
  "Une synthèse vocale destinée aux déficients visuels"
  Hélène Collavizza, Jean-Paul Stromboni
  XXVIèmes Journées d'Etude sur la Parole, 12-16 juin 2006, Dinard, (poster)
- Le CD-ROM 2007 ainsi que la synthèse vocale ont concouru au «Challenge Handicap Inter-universitaire de Metz» édition 2007. Le groupe d'élèves qui nous a représentés a remporté le premier prix de la Communication autour de l'ordinateur (voir `http://challenge-ht.sciences.univ-metz.fr/article.php3?id_article=223`).
- Le CD-ROM 2007 et la synthèse vocale ont fait l'objet d'un atelier à la journée «World Usability Day at Sophia-Antipolis» le 29 novembre 2007 (`http://worldusabilityday.org/world-usability-day-sophia-antipolis-france`).
- Le projet qui a remporté le "Prix DeViNT" 2008 a été présenté à l'émission "La tête au carré" sur France Inter le 12 Juin 2008 (voir `http://sites.radiofrance.fr/franceinter/em/lateteaucarre/index.php?id=68154`). Anne-Marie Hugues (organisatrice du prix), Jean l'Herbon de Lussats (coprésident de la journée) et les élèves concepteurs du jeu ont été les invités de Mathieu Vidard pour parler des projets DeViNT et du jeu gagnant : un simulateur de ski réalisé grâce à une planche de skateboard.

– Les projets DeViNT ont été présentés dans une table ronde et dans un atelier par Sébastien Mosser et Christian Brel, tous deux élèves de Polytech'Nice Sophia-Antipolis, aux neuvièmes "Rencontres Mondiales du Logiciel Libre" à Mont-de-Marsan en juillet 2008.

Chapitre IV

Activités de recherche

1 Présentation générale

Depuis ma nomination comme Maître de Conférences en Octobre 1992, j'ai travaillé successivement sur la vérification formelle des microprocesseurs, les contraintes sur domaines continus et la vérification des programmes.

Le thème des recherches de ma thèse de doctorat, effectuée au *Laboratoire d'Informatique de l'Université de Provence* à Marseille était la **vérification formelle des microprocesseurs**. Plus précisément, j'ai proposé une modélisation fonctionnelle du niveau d'abstraction correspondant aux *micro-séquences*, c'est-à-dire le niveau qui correspond aux *étages* dans une architecture *pipeline*. J'ai proposé une méthode de preuve basée sur la *réécriture* afin de vérifier que le niveau *micro-séquences* réalise correctement le niveau d'abstraction supérieur, qui est ici le niveau des *instructions assembleur*.

Lors de ma nomination à Nice en 1992, j'ai intégré la nouvelle équipe que le Professeur Jacques Chazarain était en train de constituer au sein du laboratoire I3S. Cette équipe incluait en particulier le Professeur Emmanuel Kounalis, qui travaillait sur la réécriture et plus précisément sur la définition d'un principe d'induction, les *test sets*. L'objectif de l'équipe était d'explorer différentes techniques de preuve, la vérification des processeurs étant un domaine d'application de ces techniques. J'ai travaillé sur la **vérification formelle des microprocesseurs** pendant cinq ans, en co-encadrant avec Jacques Chazarain le DEA puis la thèse de Laurent Arditi. L'originalité de notre approche a été de proposer une modélisation objet et une approche coopérative combinant différentes techniques de preuve. Pendant cette période, j'ai également travaillé avec Emmanuel Kounalis en co-encadrant le DEA de Ould Ahmedou Mohamed Lemine sur la généralisation de théorèmes dans les preuves inductives.

A l'issue de cette période, et suite à la réorganisation de l'équipe[1], j'ai changé de thème de recherche en travaillant avec le Professeur Michel Rueher sur les **contraintes en domaines continus**. Cette thématique m'était totalement inconnue, mais il s'est avéré par la suite que mes compétences en mathématiques numériques, ont été

[1] après le départ à la retraite de Jacques Chazarain

des bases précieuses pour me permettre de comprendre et de comparer différentes techniques de résolution de contraintes en domaines continus, qui sont pour certaines proches du calcul numérique. J'ai donc intégré l'équipe de Michel Rueher et co-encadré avec lui le DEA puis la thèse de François Delobel sur la résolution de contraintes en domaines continus.

Tout en restant dans le thème de la résolution de contraintes, mais cette fois en domaines finis, j'ai ensuite réorienté mes recherches sur la **vérification de programmes**, dans le cadre du projet RNTL DANOCOPS «Détection Automatique de Non Conformités d'un Programme vis à vis de ses Spécifications». Des travaux antérieurs d'Arnaud Gotlieb et Michel Rueher avaient montré la pertinence de l'utilisation des contraintes pour représenter un programme et trouver un jeu de test permettant d'atteindre un point dans ce programme. J'ai repris ces idées de fond, et mettant à profit mes connaissances du domaine de la preuve de matériel, j'ai proposé une méthodologie de modélisation avec abstraction booléenne de la partie contrôle du programme, qui combine résolution SAT et résolution de contraintes sur domaines finis. Nous avons ensuite amélioré cette approche en collaboration avec le Professeur Pascal van Hentenryck de l'Université de Brown, en construisant à la volée le système de contraintes numériques lors de l'exploration du graphe de flot de contrôle du programme. J'ai obtenu une délégation CNRS de dix-huit mois pour travailler sur ce thème de recherche (de Septembre 2007 à Février 2009).

Enfin, j'ai été invitée pendant six mois dans l'équipe du Professeur Mike Gordon à l'Université de Cambridge (UK). J'ai pu y explorer une autre méthode de preuve, fondée sur la logique d'ordre supérieur, en utilisant le démonstrateur de théorèmes HOL4. En collaboration avec Mike Gordon, nous avons défini une méthodologie de vérification qui reprend l'exploration à la volée du graphe de flot de contrôle du programme, et qui combine un solveur de contraintes, un solveur SMT et le démonstrateur de théorèmes HOL4. Le point fort de cette approche est qu'elle repose sur une sémantique du langage qui a été formellement définie dans HOL4. Ainsi, l'exécution symbolique d'une instruction dans le chemin en cours d'exploration, est effectuée par réduction automatique de la sémantique opérationnelle définie dans HOL4.

2 Axes de recherche

Les thèmes de recherche sur lesquels j'ai travaillé sont présentés en ordre chronologique inverse.

2.1 Détection de non conformités aux spécifications dans un programme Java

J'ai commencé à travailler sur ce thème de recherche dans le cadre du projet RNTL DANOCOPS. Il s'agissait d'étudier l'usage des techniques de résolution de contraintes pour la preuve de spécifications JML de programmes Java. Plus précisément, mettant à profit mes connaissances du domaine de la preuve de matériel, j'ai défini une méthodologie de modélisation avec abstraction booléenne de la partie contrôle du programme (l'abstraction booléenne est un passage obligé pour les outils

de type SAT utilisés pour le matériel) qui combine une résolution booléenne avec la résolution sur les entiers.

Cette première approche a été améliorée en construisant à la volée le système de contraintes, lors du parcours du graphe de flot de contrôle du programme (travaux en collaboration avec le professeur Pascal Van Hentenryck de l'Université de Brown). Le système de contraintes initial contient la pré-condition. Pour chaque instruction simple, la contrainte correspondante est ajoutée. Pour une instruction de contrôle («if then» ou «while») la condition est temporairement ajoutée au système de contraintes. Si cette condition est consistante avec le système, alors l'exécution du if ou du while est effectuée. Sinon, la contrainte est enlevée du système, sa négation est ajoutée, et l'exécution continue sur la partie « else » (ou sur l'instruction après le while). L'avantage de cette méthode est que seuls les chemins sémantiquement faisables sont vérifiés.

Mes travaux actuels portent sur deux aspects. D'une part, l'intégration du démonstrateur de théorèmes HOL4 à cette méthode de preuve. La sémantique opérationnelle du langage de programmation a été formellement définie dans HOL4, et le calcul du nouvel état est effectué par réduction mécanique de cette sémantique formelle. Cela assure que le système de contraintes construit est conforme à la sémantique opérationnelle du langage. Ce travail est effectué en collaboration avec le professeur Mike Gordon de l'université de Cambridge (UK).

D'autre part, dans le cadre du projet RNTL "TESTEC", nous travaillons avec Michel Rueher et Le-Vinh Nguyen (dont je co-encadre la thèse de doctorat) sur la vérification des programmes temps réels, en utilisant le langage de programmation par contraintes "COMET".

2.2 Contraintes sur domaines continus

En 1995 j'ai effectué une conversion thématique en travaillant avec Michel Rueher dans le domaine de la résolution de contraintes sur domaines continus. Nous avons tout d'abord établi une comparaison fine de différentes consistances partielles pour le calcul d'intervalles. Nous avons également étudié comment étendre les domaines d'un système initialement consistant. J'ai co-encadré avec Michel Rueher le DEA et la thèse de doctorat de François Delobel, sur ces thèmes de recherche. Je me suis également intéressée à l'application des techniques de résolution de contraintes en domaines continus dans le cadre de l'estimation de signaux médicaux. Ces travaux ont été effectués en collaboration avec O. Meste et Michel Rueher dans le cadre du stage de DEA de Vincent Gay-Para.

2.3 Vérification formelle des systèmes digitaux

Ce thème a été le sujet de ma thèse de doctorat et a été poursuivi lors de ma nomination à l'ESSI, en co-encadrant avec J. Chazarain le DEA puis la thèse de L. Arditi (de 1992 à 1996). Le but est la conception de circuits numériques valides, par une méthode de vérification sûre et applicable à des cas de complexité réelle. Plus précisément nous avons abordé le problème de la vérification formelle des processeurs. Le comportement des processeurs est décrit à des niveaux d'abstraction

successifs, avec une méthodologie orientée objet, et la preuve est effectuée en mettant en relation les formules obtenues à des niveaux adjacents, par des techniques de calcul formel et d'induction.

2.4 Preuves inductives

J'ai travaillé en collaboration avec E.Kounalis sur la généralisation de théorèmes dans le cadre des preuves inductives. Une des limitations des preuves par induction est que l'hypothèse d'induction ne peut pas toujours être utilisée à cause des valeurs constantes qu'elle contient. Ces travaux ont été le sujet du stage de DEA de Ould Mohamed Lemine que j'ai co-encadré avec E. Kounalis en 1995.

2.5 Activités diverses

J'ai été invitée dans l'équipe du professeur Mike Gordon à Cambridge (voir `http://www.cl.cam.ac.uk/~mjcg`) de Février 2008 à Août 2008.

J'ai été relectrice pour les conférences internationales STACS (Symposium on Theoretical Aspects of Computer Science) en 1994, CHARME (Correct HARdware design and verification MEthods) qui est devenue FMCAD (Formal Methods in Computer-Aided Design), CP (Constraint Programming) et CPAIOR (Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems).

J'ai participé aux projets RNTL DANOCOPS et TESTEC et effectué dans ce cadre plusieurs exposés.

J'ai présenté mes travaux sur la vérification des programmes par programmation par contraintes dans des séminaires à Grenoble en Janvier 2008, à l'Université de Cambridge dans le cadre du ARG lunch (Automated Reasoning Group) le 11 mars 2008 (voir `http://www.talks.cam.ac.uk/`) et enfin au laboratoire LIFC à Besançon en Novembre 2008.

J'ai été invitée au deuxième "Franco-Japanese workshop on Constraint Programming" en 2005.

3 Encadrement de thèses et de DEA

3.1 Encadrement de thèses

Co-encadrement avec Jacques Chazarain de la thèse de doctorat de l'Université de Nice Sophia-Antipolis de Laurent Arditi, sur le thème "Spécification et Preuve des Microprocesseurs", soutenue en Octobre 1996 (Mention Très Honorable, avec Félicitations du Jury).

Co-encadrement avec Michel Rueher de la thèse de doctorat de l'Université de Nice Sophia-Antipolis de François Delobel, sur le thème "Résolution de contraintes disjonctives", soutenue en Janvier 2000.

Co-encadrement avec Michel Rueher de la thèse de doctorat de l'Université de Nice Sophia-Antipolis de Le-Vinh Nguyen, sur le thème "Vérification de programmes temps réel par programmation par contraintes", commencée en avril 2008.

3.2 Encadrement de projets de Master

En 2002/2003, co-encadrement avec M. Rueher et O. Lhomme (société Ilog) d'un projet puis stage de fin d'études de K. Poupon, sur l'utilisation des contraintes pour la gestion des emplois du temps de l'ESSI.

En 2006/2007, co-encadrement avec Michel Rueher et Claude Michel du projet de master recherche PLMT de Lydie Blanchet : «Etude du couplage d'un solveur SAT et d'un solveur CSP pour la vérification de programmes».

En 2007/2008, encadrement du projet de master professionnel PLMT de Sébastien Derrien et Eric Le Duff : «JPPView : un plug-in eclipse de visualisation du processus de validation d'un programme Java vis à vis d'une spécification JML».

3.3 Encadrement de DEA

En 1997, co-encadrement avec Michel Rueher et Olivier Meste du projet de DEA de Vincent Gay-Para sur l'application des techniques de résolution de contraintes à l'optimisation de problèmes de traitement de signaux biomédicaux.

En 1996, co-encadrement avec Michel Rueher du projet de DEA de François Delobel sur la résolution des contraintes disjonctives.

En 1995, co-encadrement avec Emmanuel Kounalis du projet de DEA d'Ould Mohamed Lemine sur la généralisation de théorèmes dans les preuves inductives.

En 1993, co-encadrement avec Jacques Chazarain du projet de DEA de Laurent Arditi intitulé : "S.V.P : Spécification et Vérification de processeurs, environnement interactif pour la Spécification et la Vérification formelle des Microprocesseurs".

Chapitre V

Publications

Les publications effectuées durant mon doctorat sont désignées par une étoile *

1 Revues internationales

[1] H. Collavizza, M. Rueher, P. Van Hentenryck. Constraint-Programming for Bounded Program Verification. En cours de révision pour *Constraints, an International Journal*, 2009.

[2] H. Collavizza, F. Delobel, M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, Kluwer Academic Publishers, Vol.5(3), pp. 213-228, 1999.

2 Revues nationales

[3] L. Arditi, H. Collavizza. Intégration de techniques coopératives pour la vérification formelle des processeurs. Numéro spécial "Méthodes formelles : validation de systèmes complexes", *Technique et Science Informatique*, n°6/97, 1997.

[4*] D. Borrione, J.L. Paillet, L. Pierre, H. Collavizza. Modélisation fonctionnelle et preuve de circuits digitaux. *Technique et Science Informatique*, Vol. 8, N°6, 1989.

3 Chapitre d'ouvrage

[5] M.J.C. Gordon, H. Collavizza. Forward with Hoare. *Reflections on the Work of C.A.R. Hoare.* Accepté pour publication, à paraître dans History of Computing Series, Springer.

4 Conférences d'audience internationale avec comité de sélection

[6] H. Collavizza, M. Rueher, P. Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. *12th CP'2008*, Sidney, *LNCS 5202*, pp. 327-341, 2008, Springer-Verlag.

[7] H. Collavizza, M. Rueher. Exploring different constraint-based modelings for program verification, *13th CP'2007*, *LNCS 4741*, Providence, September 25-29, 2007.

[8] H. Collavizza, M. Rueher. Exploration of the constraint programming technique capabilities in the software verification process. *TACAS'2006*, *LNCS 3920*, pp. 182-196, 2006.

[9] H. Collavizza, J.P. Stromboni. Une synthèse vocale destinée aux déficients visuels. *XXVIèmes Journées d'Etude sur la Parole*, 12-16 juin 2006, Dinard (poster).

[10] H. Collavizza, F. Delobel, M. Rueher. Extending consistent domains of numeric CSP. *IJCAI'99*, Stockholm, Sweden, 31 July - 6 August 1999.

[11] H. Collavizza, F. Delobel, M. Rueher. A Note on Partial Consistencies over Continuous Domains Solving Techniques. *CP'98 (Fourth International Conference on Principles and Practice of Constraint Programming)*, Pisa, Italy, October 26-30, 1998, *LNCS 1520* (Springer Verlag), pp. 147-161.

[12] L. Arditi, H. Collavizza Towards verifying VHDL descriptions of Processors *EURODAC'95 with EURO-VHDL IEEE int. Conf.*, IEEE Comp. Society Press, pp 414-419, Brighton, Sept. 1995.

[13] L. Arditi, H. Collavizza. An Object-oriented Framework for the Formal Verification of Processors *ECOOP "European Conference on Object Oriented Programming"*, Ed. Olthoff, *LNCS 952*, pp 213-234, Aarhus, Danemark, Aout 95.

[14] J. Chazarain, H. Collavizza. Combining Symbolic Evaluation and Object-oriented Approach for Verifying Processor-like Architectures at the RT-level *IFIP WG10.2 Advanced Research Conference*, *LNCS 683*, pp 109-121, CHARME'93, Arles, Mai 1993.

[15*] H. Collavizza. μSPEED : a System for the Specification and the Verification of Micro-processors. *9th Symposium on Theoretical Aspects of Computer Science*, Cachan, *LNCS 577*, Fev. 1992 (poster).

[16*] H. Collavizza. Functional Semantics of Microprocessors at the Micro-program level and Correspondence with the Machine Instruction level *IEEE European Design Automation Conference*, Gordon Adshead and Jochen A. G. Jess editors, IEEE Computer Society, ISBN 0-8186-2024-2, 12-15 March, Glasgow, 1990.

5 Workshops internationaux avec comité de sélection

[17*] D. Borrione, H. Collavizza, C. Le Faou. μSPEED : a Framework for Specifying and Verifying Micro-processors. *ACM, Int. Workshop on Formal Methods in VLSI Design*, Miami (USA), 1991.

[18*] H. Collavizza, D. Borrione. Specifying and Verifying the Micro-program Parallelism in Micro-processors of the Von Neumann type" *Int. Workshop on Designing Correct Circuits*, Oxford 26-28 Sept. 1990. G. Jones. M. Sheeran ed. Springer Verlag ISBN 3.540.196959.5.

6 Rapports de Recherche et communications dans des workshops

[19] Semantically-driven Bounded Model Checking using Theorem Proving, SMT and Constraint Solving. Hélène Collavizza, Mike Gordon. Rapport de Recherche I3S/RR-2009-13-FR, Septembre 2009, 17 pages.

[20] Integration of Theorem-proving and Constraint Programming for Software Verification. Hélène Collavizza, Mike Gordon. Rapport de Recherche I3S/RR-2008-21-FR, Novembre 2008, 17 pages.

[21] Comparison between CPBPV with ESC/Java, CBMC, Blast, EUREKA and Why. Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Rapport de Recherche I3S, I3S/RR-2008-06-FR, Juin 2008.

[22] CPBPV : A Constraint-Programming Framework For Bounded Program Verification. H. Collavizza, Michel Rueher, Pascal van Hentenryck. RR IRSN I3S/RR 2008-04-FR. Feb 2008.

[23] Exploration of the capabilities of constraint programming for software verification. H. Collavizza. FJCP'2005, 2nd Franco-Japanese workshop on Constraint Programming, Le Croisic, France, nov. 2005.

[24] A Framework for Systematic Specification and Efficient Verification of Processors. L. Arditi, H. Collavizza. Rapport de Recherche n°95-06, Université de Nice - Sophia Antipolis, CNRS URA 1376, Fev. 1995.

[25] Equational Logic : Overview and Proof methods. E. Kounalis, L. Arditi, J. Chazarain, H. Collavizza, P. Collet, L. Mallé, S. Muller, C. Solnon. RR n°94-08, Université de Nice - Sophia Antipolis, CNRS URA 1376, Février 1994.

Résumé : Ce mémoire d'Habilitation à Diriger des Recherches présente mes contributions à la vérification formelle des processeurs et des programmes, ainsi qu'à la programmation par contraintes. La vérification formelle, tant de matériel que de logiciel, est cruciale pour la sécurité des systèmes critiques, est un enjeu économique important et reste un défi pour la recherche.

Les méthodes de vérification formelle retenues, aussi bien pour la vérification des processeurs que des programmes sont des méthodes entièrement automatiques qui reposent sur l'utilisation de procédures de décision. Pour la vérification de programmes, la résolution de contraintes sur domaines finis fournit une procédure de décision sur les entiers bornés (codables en machine). L'explosion combinatoire est retardée par la combinaison de solveurs spécifiques (booléen, linéaires, domaines finis), ce qui a permis d'obtenir des résultats expérimentaux qui surpassent dans certains cas les outils de "bounded model checking" basés sur l'utilisation de solveurs SAT.

Dans un second temps, la vérification formelle des programmes est également abordée sous l'angle du développement conjoint d'une vérification complète et d'une exploration par model checking, basés sur la sémantique formelle du langage définie dans l'assistant de preuves HOL4.

Enfin, ce mémoire présente mes contributions sur les contraintes en domaines continus (i.e. où les variables sont des nombres réels). Ces contraintes ont de nombreuses applications pratiques, par exemple en mécanique ou avionique, et leurs mécanismes de résolution peuvent servir de base à la vérification de programmes en présence de nombres flottants.

Summary : This habilitation thesis presents my contributions to the formal verification of processors and programs, and to constraint programming. Formal verification of hardware and software is crucial for the safety of critical systems, is an important economic issue and remains a challenge for research.

The formal methods we explored for the verification of processors and programs are entirely automatic and based on decision procedures. For the formal verification of programs, the resolution of constraints on finite domains provides a decision procedure on bounded integers (i.e. machine-codable). The combinatorial explosion is delayed by the combination of specific solvers (Boolean, linear, finite domains). This has made possible to obtain experimental results outperforming in some cases state of the art bounded model checkers based on SAT solvers.

In a second step, the formal verification of programs is also approached under the angle of the joint development of a complete proof and an exploration by model checking. Both complete proof and model checking are based on the formal semantics of the language defined in the proof assistant HOL4.

Lastly, this habilitation thesis presents my contributions on numerical constraints (i.e where variables are real numbers). These constraints have many practical applications, for example in mechanics or avionics. Furthermore, their resolution mechanisms can be a basis for the formal verification of programs with floating point numbers.